Attorney Docket No. 1075.1174

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent Application of:

Akio MATSUDA, et al.

Application No.: Group Art Unit:

Filed: September 28, 2001 Examiner:

For: METHOD OF SIMULATING OPERATION OF LOGICAL UNIT, AND COMPUTER-READABLE RECORDING MEDIUM RETAINING PROGRAM FOR SIMULATING OPERATION OF LOGICAL UNIT

SUBMISSION OF CERTIFIED COPY OF PRIOR FOREIGN APPLICATION IN ACCORDANCE WITH THE REQUIREMENTS OF 37 C.F.R. § 1.55

Assistant Commissioner for Patents
Washington, D.C. 20231

Sir:

In accordance with the provisions of 37 C.F.R. § 1.55, the applicant(s) submit(s) herewith a certified copy of the following foreign application:

Japanese Patent Application No. 2000-403135

Filed: December 28, 2000

It is respectfully requested that the applicant(s) be given the benefit of the foreign filing date(s) as evidenced by the certified papers attached hereto, in accordance with the requirements of 35 U.S.C. § 119.

Respectfully submitted,

STAAS & HALSEY LLP

Date: September 28, 2001 By: _____

James D. Halsey, Jr.
Registration No. 22,729

700 11th Street, N.W., Ste. 500
Washington, D.C. 20001
(202) 434-1500

日　本　国　特　許　庁
JAPAN PATENT OFFICE

別紙添付の書類に記載されている事項は下記の出願書類に記載されている事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed with this Office.

出願年月日
Date of Application: ２０００年１２月２８日

出願番号
Application Number: 特願２０００－４０３１３５

出願人
Applicant(s): 富士通株式会社

２００１年　６月２０日

特許庁長官
Commissioner,
Japan Patent Office 及川耕造

出証番号　出証特２００１－３０５８４６２

【書類名】　特許願

【整理番号】　0051937

【提出日】　平成12年12月28日

【あて先】　特許庁長官殿

【国際特許分類】　G06F 19/00

【発明の名称】　論理装置の動作シミュレーション方法並びに論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体

【請求項の数】　10

【発明者】

【住所又は居所】　神奈川県川崎市中原区上小田中４丁目１番１号　富士通株式会社内

【氏名】　松田　明男

【発明者】

【住所又は居所】　神奈川県川崎市中原区上小田中４丁目１番１号　富士通株式会社内

【氏名】　朱　強

【発明者】

【住所又は居所】　神奈川県川崎市中原区上小田中４丁目１番１号　富士通株式会社内

【氏名】　松崎　和浩

【発明者】

【住所又は居所】　神奈川県川崎市中原区上小田中４丁目１番１号　富士通株式会社内

【氏名】　土居　武史

【特許出願人】

【識別番号】　000005223

【氏名又は名称】　富士通株式会社

【代理人】

【識別番号】　100092978

【弁理士】

【氏名又は名称】　真田　有

【電話番号】　0422-21-4222

【手数料の表示】

【予納台帳番号】　007696

【納付金額】　21,000円

【提出物件の目録】

【物件名】　明細書　1

【物件名】　図面　1

【物件名】　要約書　1

【包括委任状番号】　9704824

【プルーフの要否】　要

【書類名】　　明細書

【発明の名称】　論理装置の動作シミュレーション方法並びに論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体

【特許請求の範囲】

【請求項１】　プログラムの実行単位であるスレッドの制御を行なうスレッドマネージャが、論理装置の設計仕様に応じて該論理装置の動作完了までに必要となる機能を表したスレッドの実行に必要なハードウェアリソースを、当該ハードウェアリソースを管理するリソースマネージャに要求するリソース要求ステップと、

該リソースマネージャが、該要求に応じたハードウェアリソースを予め規定されたルールに従って該スレッドに割り当てるリソース割り当てステップと、

該スレッドマネージャが、該リソースマネージャによる割り当て結果に応じて該スレッドの実行状態を制御するスレッド制御ステップとを有するとともに、

該スレッドの実行が完了するまで該スレッドマネージャと該リソースマネージャとが連携して上記の各ステップを繰り返し実行することにより、該論理装置の動作完了までの動作をシミュレーションすることを特徴とする、論理装置の動作シミュレーション方法。

【請求項２】　該リソースマネージャが、該リソース要求ステップによるリソース要求を監視し、その監視結果に基づいて複数スレッド間のリソース要求のデッドロック状態を判定することを特徴とする、請求項１記載の論理装置の動作シミュレーション方法。

【請求項３】　該リソースマネージャが、該リソース要求ステップによるリソース要求によって割り当てられたハードウェアリソースに対するリード／ライト要求を監視し、その監視結果に基づいて複数スレッドのハードウェアリソースに対するリード／ライト動作の競合状態を判定することを特徴とする、請求項１記載の論理装置の動作シミュレーション方法。

【請求項４】　該リソースマネージャが、該ハードウェアリソースに対するリソース要求数を監視し、その監視結果に基づいて該スレッドのボトルネックを検出することを特徴とする、請求項１記載の論理装置の動作シミュレーション方

法。

【請求項５】　該リソースマネージャが、該ハードウェアリソースに対するリソース要求数を監視し、その監視結果に基づいて該リソース要求のブロッキングを検出することを特徴とする、請求項１記載の論理装置の動作シミュレーション方法。

【請求項６】　該スレッドに、該リソースマネージャによって割り当てられたハードウェアリソースを占有する時間についての予算を与えておくことを特徴とする、請求項１記載の論理装置の動作シミュレーション方法。

【請求項７】　該スレッドに、該機能の実行制限時間を与えておくことを特徴とする、請求項１記載の論理装置の動作シミュレーション方法。

【請求項８】　請求項１～７のいずれか１項に記載の動作シミュレーション方法によるシミュレーション結果と、該論理装置の動作予測値とを比較する比較ステップと、

該比較ステップでの比較結果を外部装置へ出力する出力ステップとを有することを特徴とする、論理装置の動作シミュレーション方法。

【請求項９】　論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体であって、

該コンピュータを、

プログラムの実行単位であるスレッドの制御を行なうスレッドマネージャ及び該スレッドの実行に必要なハードウェアリソースを管理するリソースマネージャとして機能させるとともに、

該スレッドマネージャが、論理装置の設計仕様に応じて該論理装置の動作完了までに必要となる機能を表したスレッドの実行に必要なハードウェアリソースを該リソースマネージャに要求するリソース要求ステップと、該リソースマネージャが、該要求に応じたハードウェアリソースを予め規定されたルールに従って該スレッドに割り当てるリソース割り当てステップと、該スレッドマネージャが、該リソースマネージャによる割り当て結果に応じて該スレッドの実行状態を制御するスレッド制御ステップとを実行するとともに、該スレッドの実行が完了するまで該スレッドマネージャと該リソースマネージャとが連携して上記の各ステッ

プを繰り返し実行することにより、該論理装置の動作完了までの動作をシミュレーションするためのプログラムが記録されていることを特徴とする、論理装置の動作シミュレーションプログラムを記録した記録媒体。

【請求項１０】　論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体であって、

該コンピュータに、

請求項１～７のいずれか１項に記載の動作シミュレーション方法によるシミュレーション結果と該論理装置の動作予測値とを比較する比較ステップと、該比較ステップでの比較結果を外部装置へ出力する出力ステップとを実行させるための動作シミュレーションプログラムが記録されたことを特徴とする、論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】

本発明は、ＬＳＩ（Large Scale Integrated circuit）やＩＣ（Integrated Circuit）などの論理装置を設計する上でその機能や性能の検証を目的として動作シミュレーションを行なうのに用いて好適な、論理装置の動作シミュレーション方法並びに論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体に関する。

【０００２】

【従来の技術】

従来、論理装置の設計はＨＤＬ(Hardware Description Language)を用いてＲＴ（Register Transfer）レベルから設計を開始することが多い。しかし、ＲＴレベルはクロックサイクル精度のハードウェアを意図した技術なので、ハードウェアとソフトウェアが明確に分離していない設計初期段階の論理装置を表現して機能を検証したり性能を評価したりすることは困難である。

【０００３】

そこで、ＲＴレベルよりも抽象度の高い設計の初期段階で論理装置を記述する

言語〔例：Ｃ／Ｃ++，ＵＭＬ（Unified Modeling Language），ＳＤＬ(Specification and Description Language)など〕を用いる設計手法が数多く提案されている。例えば、以下に列記する文献がある。

①「Hardware-Software Co-design of Embedded Systems - The POLIS approach」（F.Balarin, E.Sentovich, M.Chiodo, P.Giusto, H.Hsieh, B.Tabbara and A.Jurecska, L.Lavagno, C.Passerone, K.Suzuki and A.Sangiovanni-VinCentelli, Kluwer Academic Publishers, 1997.）

②「Coware - a design environment for heterogeneous hardware/software partitioning problem」（K.Rompaey, D.Verkest, I.Bolsens and H.De Man, Proceedings EuroDAC, 1996.）

③「An Object Oriented Programming Approach for Hardware Design」（S.Vernalde, P.Schaumont and I. Bolsens, IEEE Computer Society Workshop on VLSI, April, 1999.）

④「A Methodology and Design Environment for DSP ASIC Fixed Point Refinement」（R.Cmar, L.Rijnders, P.Schaumount, S.Vernalde and I.Bolsens, Proceedings Design and Test in Europe Conference, pp. 271-276, March, 1999.）

⑤「Hardware Reuse at the Behavioral Level」（P.Schaumont, R.Cmar, S.Vernalde, M.Engels and I.Bolsens, The proceedings of 36th Design Automation Conference. ACM, 1999, pp. 784-789, June 1999.）

⑥「An Efficient Implementation of Reactivity for Modeling Hardware in the SCENIC Design Environment」（Stan Liao, Steve Tijang and Rajesh Gupta, Proceedings of the Design Automation Conference DAC'97, pp. 70-75, June 1999.）

⑦「SpecC System-Level Design Methodology Applied to the Design of a GSM Vocoder」（A.Gerstlauer, S.Zhao, A.Horak and D.Gajski, Proceedings of the Workshop on Synthesis And System Integration of MIxed Technologies, April, 2000.）

⑧「On the use of C++ for system-on-chip design」（Diederik Verkest, Joh

an Cockx and Freddy Potargent, Proceedings of the IEEE Workshop on VLSI (IWV), pp.42-47, April, 1999.)

【０００４】

これらの設計手法では論理装置の並列／並行処理を実現するためにソフトウェアのスレッド技術〔参考文献「並列オペレーティングシステム」（福田晃，(株)コロナ社，pp.30，1997.），「Ｊａｖａスレッドプログラミング」（Scott Oaks，Henry Wong共著，戸松豊和，西村利浩訳，（株）オーム社，1997.）など〕を導入している。

【０００５】

即ち、各スレッドを１つの処理ブロックにマッピングして、ブロック間には「通信」機能を定義することで論理装置のシミュレーションを実現している。

ここで、論理装置の設計コストに影響する大きな要素は二つ存在している。１つは手戻り作業の有無であり、もう１つは設計の再利用性である。

即ち、手戻り作業が多いほど、設計コストは増加する。特に、設計仕様書（以下、単に「仕様」という）からいきなりＨＤＬを用いてＲＴレベルの設計を始める場合に、仕様の確認を行なわず、実装設計を開始すると、実装した結果が要求されている仕様を満たさない場合に、仕様の見直しが必要となり、実装設計の再設計を行なわなければならないケースが多い。これは設計工数、設計期間を大きく増大させる原因である。これを改善するために、早期に正確なパフォーマンスの見積もりを行なうことが重要である。

【０００６】

もう一方、設計の再利用性に関しては、二種類の再利用の可能性があげられる。一つは以前設計したモジュールの再利用であり、もう一つは設計を段階化したときの途中段階の記述（ソースコード）の再利用である。例えば、設計の途中からパフォーマンスを見積もった結果によって、再設計になった場合に、ソースコードを変更しなければならないことは設計コストに影響する。この視点から、仕様書から直接ＲＴレベルを設計する手法は設計コストが高い。なぜなら、設計を変更すると、最悪の場合、全てのＲＴレベルでの記述を再記述しなければならないことがあるからである。

【０００７】

以上のように、仕様から直接ＲＴレベルで設計プロセスを開始することは設計コストの増大につながることがわかる。そこで、これを解決するために、システムの上位設計の手法が提案されている。例えば図３０（Ａ）及び図３０（Ｂ）に示すような、段階的詳細化による設計アプローチである。

この段階的詳細化による設計アプローチは、概略して、以下のような手順によって設計が行なわれる。

【０００８】

・仕様１００から機能（ブロック）を抽出してブロック図化する〔図３０（Ａ）のステップＡ１〕。そして、そのブロック図に基づいて、例えば、Ｃ言語などによって、「UnTimed」レベルの機能モデル、つまり、時間概念を考慮しない機能モデルを構築（記述）する〔図３０（Ａ）のステップＡ２，図３０（Ｂ）のステップＢ１〕。なお、このとき、機能（ブロック）間に依存関係があれば、その間には「通信」機能が定義（記述）され、それらの機能（ブロック）はシーケンシャルに実行される。

【０００９】

・アーキテクチャの設計によって、ハードウェアにマッピングする機能ブロックの記述をＢＣＡ（Bus Cycle Accurate）に変更し〔図３０（Ｂ）のステップＢ２〕、「通信」機能はバス通信プロトコルになる〔図３０（Ｂ）のステップＢ３〕。この段階（レベル）でパフォーマンスの見積もりを行ない〔図３０（Ａ）のステップＡ３〕、仕様の確認を行なう。

【００１０】

・さらに記述をＦＣＡ（Full Cycle Accurate）に変更して〔図３０（Ｂ）のステップＢ４〕、実際のＲＴレベルの記述に変更する〔図３０（Ａ）のステップＡ４〕。このとき、「通信」機能も記述に組み込む〔図３０（Ａ）のステップＡ５〕。

以上のような段階的な詳細化によって、早期に仕様の確認ができ、実行可能な仕様を実現することができる。

【００１１】

【発明が解決しようとする課題】

しかしながら、上述したような従来手法では、各レベルでの変更を全て手作業で行なうため、上記の「UnTimed」レベルからＢＣＡレベルに変更する際に余計な設計コストが必要となる。さらに、その段階でのパフォーマンスの見積もり（上記のステップＡ３）において、設計の見直しが必要になった場合、記述を変更しなければならない。

【００１２】

また、従来手法では、機能（ブロック）の構成にアーキテクチャが反映されているため、アーキテクチャの変更が必要になった場合、その機能（ブロック）の構成を変更しなければならず、これに伴ってその記述も変更しなければならなくなる。

【００１３】

例えば図３１に示すように、ＢＣＡ記述により、ハードウェア資源（リソース）「Ａ」を用いて機能「１」が実現（ハードウェア資源「Ａ」に機能「１」がマッピングされる）とともに、ハードウェア資源「Ｂ」を用いて機能「２」が実現され、且つ、これらの機能「１」，「２」間に「通信」機能が定義された状態で、さらに、ハードウェア資源「Ａ」を複数用いて上記と同じ機能「１」を追加実現する必要があった場合を想定する。

【００１４】

この場合、追加分の機能「１」のＢＣＡ記述をハードウェア資源「Ａ」の個数分だけ用意しなければならず、さらに、機能「１」と機能「２」との間の「通信」機能を既に定義済みの分も含めて再定義しなければならない。このように、従来手法では、機能（ブロック）の構成にアーキテクチャが反映されているため、アーキテクチャの変更が必要になる度に機能（ブロック）の記述を全て変更する必要があり、設計コストが大幅に増大し、設計効率が悪くなる。

【００１５】

本発明は、このような課題に鑑み創案されたもので、論理装置の設計の初期段階において、動作シミュレーションを行なって、機能の確認，アーキテクチャの評価を行なえるようにするとともに、アーキテクチャの変更に対しても最小限の

記述の変更で柔軟に対応できるようにすることで、論理装置の設計コストを大幅に削減できるようにすることを目的とする。

【００１６】

【課題を解決するための手段】

上記の目的を達成するために、本発明の論理装置の動作シミュレーション方法（請求項１）は、①プログラムの実行単位であるスレッドの制御を行なうスレッドマネージャが、論理装置の設計仕様に応じてその論理装置の動作完了までに必要となる機能を表したスレッドの実行に必要なハードウェアリソースを、そのハードウェアリソースを管理するリソースマネージャに要求するリソース要求ステップと、②上記のリソースマネージャが、その要求に応じたハードウェアリソースを予め規定されたルールに従って上記スレッドに割り当てるリソース割り当てステップと、③上記のスレッドマネージャが、このリソースマネージャによる割り当て結果に応じて上記スレッドの実行状態を制御するスレッド制御ステップとを有するとともに、上記スレッドの実行が完了するまで上記のスレッドマネージャとリソースマネージャとが連携して上記①～③の各ステップを繰り返し実行することにより、論理装置の動作完了までの動作をシミュレーションすることを特徴としている。

【００１７】

なお、上記のリソースマネージャは、上記のリソース要求ステップ（①）によるリソース要求を監視し、その監視結果に基づいて複数スレッド間のリソース要求のデッドロック状態を判定するようにしてもよい（請求項２）。

【００１８】

また、上記のリソースマネージャは、上記のリソース要求ステップ（①）によるリソース要求によって割り当てられたハードウェアリソースに対するリード／ライト要求を監視し、その監視結果に基づいて複数スレッドのハードウェアリソースに対するリード／ライト動作の競合状態を判定するようにしてもよい（請求項３）。

【００１９】

さらに、上記のリソースマネージャは、上記のハードウェアリソースに対する

リソース要求数を監視し、その監視結果に基づいてスレッドのボトルネックを検出するようにしてもよい（請求項４）。

また、上記のリソースマネージャは、上記のハードウェアリソースに対するリソース要求数を監視し、その監視結果に基づいてリソース要求のブロッキングを検出するようにしてもよい（請求項５）。

【００２０】

さらに、上記のスレッドには、上記のリソースマネージャによって割り当てられたハードウェアリソースを占有する時間についての予算を与えておいてもよいし（請求項６）、上記の機能に実行制限時間を与えておいてもよい（請求項７）。

また、本発明の論理装置の動作シミュレーション方法（請求項８）は、上記の動作シミュレーション方法によるシミュレーション結果と、上記論理装置の動作予測値とを比較する比較ステップと、この比較ステップでの比較結果を外部装置へ出力する出力ステップとを有していることを特徴としている。

【００２１】

次に、本発明の論理装置の動作シミュレーションプログラムを記録した記録媒体（請求項９）は、論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体であって、上記のコンピュータを、プログラムの実行単位であるスレッドの制御を行なうスレッドマネージャ及び上記スレッドの実行に必要なハードウェアリソースを管理するリソースマネージャとして機能させるとともに、①上記のスレッドマネージャが、論理装置の設計仕様に応じて該論理装置の動作完了までに必要となる機能を表したスレッドの実行に必要なハードウェアリソースを上記のリソースマネージャに要求するリソース要求ステップと、②上記のリソースマネージャが、その要求に応じたハードウェアリソースを予め規定されたルールに従って上記スレッドに割り当てるリソース割り当てステップと、③上記のスレッドマネージャが、上記のリソースマネージャによる割り当て結果に応じて上記スレッドの実行状態を制御するスレッド制御ステップとを実行するとともに、上記スレッドの実行が完了するまで上記のスレッドマネージャとリソースマネージャとが連携して上記①～③の各ステップを繰り返し実行

することにより、論理装置の動作完了までの動作をシミュレーションするためのプログラムが記録されていることを特徴としている。

【００２２】

さらに、本発明の論理装置の動作シミュレーションプログラムを記録した記録媒体（請求項１０）は、コンピュータに、上記の動作シミュレーション方法によるシミュレーション結果と論理装置の動作予測値とを比較する比較ステップと、この比較ステップでの比較結果を外部装置へ出力する出力ステップとを実行させるための動作シミュレーションプログラムが記録されたことを特徴としている。

【００２３】

【発明の実施の形態】

以下、図面を参照して本発明の実施の形態を説明する。

（Ａ）第１実施形態の説明

図１は本発明の第１実施形態に係るパーソナルコンピュータなどの計算機（情報処理装置）の構成を示すブロック図で、この図１に示す計算機１は、計算機本体２とディスプレイ（表示装置）３とをそなえており、計算機本体２（以下、単に「本体２」と略記することがある）には、例えば、ＣＰＵ（Central Processing Unit）４，主記憶部（メモリ）５，二次記憶装置（ハードディスク）６，フロッピーディスク（ＦＤ）ドライブ７などがそなえられており、これらのコンポーネントがＰＣＩ（Peripheral Component Interconnect）バスなどの内部バス８を介して相互に通信可能に接続されている。

【００２４】

ここで、上記のＣＰＵ４は、計算機１としての動作を統括制御するためのもので、メモリ５やハードディスク６に内部バス８を介してアクセスして必要なソフトウェア（アプリケーション）プログラム（以下、単に「プログラム」ともいう）やアプリケーションデータなどを読み込んで動作することによって、計算機１として必要な機能（本実施形態では、特に、論理装置の動作シミュレーション装置としての機能）が発揮されるようになっている。なお、その動作結果（論理装置の動作シミュレーション結果を含む）は、適宜、ディスプレイ３に表示（出力）される。

【００２５】

また、ハードディスク６は、上記のプログラムやアプリケーションデータなど（以下、説明の便宜上、単に「各種データ」と総称することがある）を予め、あるいは、インストールなどによって記憶しておくためのもので、ここに記憶されている各種データが、適宜、ＣＰＵ４からのアクセス速度が高速なメモリ５に読み出されて、ＣＰＵ４によるプログラムの実行が高速に行なわれるようになっている。

【００２６】

さらに、ＦＤドライブ７は、フロッピーディスク（ＦＤ；記録媒体）９に記録されている各種データをＣＰＵ４の制御のもとに読み出してハードディスク６に記憶することによって、各種データのインストールを可能にする機能を提供するもので、例えば、ＦＤ９に、論理装置の動作シミュレーションプログラム１０が記録されていれば、このＦＤ９からその動作シミュレーションプログラム１０をインストールする（ハードディスク６に記憶する）ことによって、計算機１（ＣＰＵ４）を論理装置の動作シミュレーション装置として機能させることが可能である。

【００２７】

なお、上記の論理装置の動作シミュレーションプログラム１０（以下、単に「シミュレーションプログラム１０」、もしくは、「プログラム１０」という）がハードディスク６あるいはメモリ５に記憶された時点で、そのプログラム１０を保持したハードディスク６あるいはメモリ５が上記シミュレーションプログラム１０を記録した記録媒体となることはいうまでもない。

【００２８】

また、このプログラム１０は、勿論、ＦＤ９以外の記録媒体〔例えば、ＣＤ－ＲＯＭや光磁気ディスク（ＭＯ）など〕に記録されていてもよく、対応するドライブなどが本体２に装備されていれば、上記と同様に、それらの記録媒体からのインストールが可能である。さらに、記録媒体からのインストールだけでなく、例えば、インターネットなどの所望の通信回線（ネットワーク）を介したオンラインでのインストールも可能である。つまり、プログラム１０は、ＦＤ９やＣＤ

－ＲＯＭ，ＭＯなどの記録媒体として提供されてもよいし、インターネットなどの所望の通信回線を介して提供されてもよい。

【００２９】

さて次に、以下では、上記のシミュレーションプログラム１０の詳細について説明する。

本実施形態のシミュレーションプログラム１０は、例えば図２に示すようなＯＭＴ（Object-oriented Modeling Technique）に基づいて記述（オブジェクト指向プログラミング）されており、その要部に着目すると、この図２に示すように、スレッドマネージャ１１，リソースマネージャ１２，スレッド１３，リソース１４，テストベンチ１５，テストデータ１６，入力キュー１７，実行待ちキュー１８，（リソース）要求１９，回答２１，リソース回答キュー２２などが定義されている。なお、「ＯＭＴ」については、例えば、文献「オブジェクト指向方法論ＯＭＴ」（J.ランボー，M.ブラハ，W.プレラニ，F.エディ，W.ローレンセン共著，羽生田栄一訳，（株）トッパン，1995.）になどに詳しく解説されている。

【００３０】

ここで、まず、テストベンチ１５は、論理装置の動作シミュレーションを行なうのに必要なテストデータ１６を所定のデータ生成ルールに従って生成してから実行権利を入力キュー１７に譲るという機能を提供するもので、このテストベンチ１５で生成されたテストデータ１６がテストベンチ１５の所有する入力キュー１７に順次格納されたのち、スレッドマネージャ１１によって順次取り出されて、スレッド１３の生成に使用されるようになっている。

【００３１】

次に、上記のスレッド１３は、「実行権利を譲る」，「実行する」，「停止する」，「停止する」，「再開する」，「開始する」，「消滅する」，「生成する」などの関数（ＯＭＴでは「メソッド」と呼ばれる）が定義されることにより、スレッドマネージャ１１から実行権利を譲り受けた場合にスレッド１３の実行状態が制御されるようになっている。

【００３２】

このスレッド１３は、プログラムの実行単位を表し、本実施形態では、実現し

たい論理装置の設計仕様に応じて必要となる論理装置の入力から出力（動作完了）までの処理（機能）を表した（定義した）逐次フローとして記述される。仮に、論理装置の入力から出力までに必要な処理（機能）を処理Ａ→処理Ｂ→処理Ｃとし、これらを１つのスレッド１３に表すとすると、例えばＣ＋＋記述スタイルを用いた場合、図５に示すように記述される。なお、以降の説明においても、シミュレーションプログラム１０の具体的な記述例については、特に断らない限り、Ｃ＋＋記述スタイルを適用するものとする。

【００３３】

そして、上記のスレッド１３は、実行状態になると、その進行（即ち、上記の処理Ａ，Ｂ，Ｃなどの「メソッド」の実行）のためにリソース１４を確保する必要がある場合はそれを要求する要求（リソース要求）１９を、確保したリソース１４を解放する場合にはそれを要求する要求（解放要求）１９を、それぞれ、自立的に発行してリソース要求キュー２０に順次格納してゆくようになっている。

【００３４】

なお、ここでいうリソース１４とは、本実施形態では、ハードウェア資源（ハードウェアリソース）に関する情報を意味し、例えば、プロセッサ，ＡＳＩＣ（Application Specific Integrated Circuit），演算器などがそれぞれリソースデータとして定義される。

【００３５】

次に、実行待ちキュー１８は、複数のキューのそれぞれに実行待ちのスレッド１３を格納する機能を提供するもので、この実行待ちキュー１８は、スレッドマネージャ１１に集約されている。

【００３６】

つまり、本実施形態のスレッドマネージャ１１は、実行待ちキュー１８と１つ以上のスレッド１３から構成され、スレッド１３の進行（生成，実行，停止，再開，消滅などの「メソッド」）及び複数のスレッド１３の実行スケジュール（順序）を管理する（即ち、スレッド１３の制御を行なう）タスクとして機能する。なお、スレッド１３の次回の動作は、このスレッドマネージャ１１が、リソースマネージャ１２の発行する回答２１をリソース回答キュー２２から取り出して参

照することで決定される（詳細については後述する）。

【００３７】

さらに、リソースマネージャ１２は、スレッド１３が進行するのに（「メソッド」の実行に）必要なリソース１４（種類及び個数）を管理し、予め規定された上記「調停ルール」に従って単位時間当たりのリソース要求１９をリソース１４の種類毎に調停しながら、リソース要求１９に応じたリソース１４を要求元のスレッド１３に割り当てるものである。

【００３８】

なお、このリソースマネージャ１２によるリソース１４の割り当て結果は回答２１として発行され、一旦、リソース回答キュー２２に格納（追加）されたのち、スレッドマネージャ１１によって取り出されて参照される。また、上記の「単位時間」とは、リソースマネージャ１２がリソース１４の調停を行なう最小の時間単位を意味する。

【００３９】

ここで、本リソースマネージャ１２を、上記のスレッド１３と同様にＣ＋＋記述スタイルで記述すると、例えば図６に示すようになる。即ち、リソースマネージャ１２が管理するリソース（Ｒ）１４の種類を例えば“Ｒ１”，“Ｒ２”とし、上記のリソース要求キュー２０とリソース回答キュー２２とを定義する。

【００４０】

そして、要求メソッド３１ａ、リソース１４の解放を行なう解放メソッド３１ｂ、リソース１４の調停を行なう調停メソッド３１ｃをそれぞれ定義する。ここで、要求メソッド３１ａは、どの種類のリソース１４（“Ｒ１”ｏｒ“Ｒ２”）を要求するのかをパラメータとして明確に指定し、要求したリソース１４とその要求１９を出したスレッド１３のスレッドＩＤとを要求メッセージとしてリソース要求キュー２０に登録する操作を行なうための関数である。

【００４１】

また、解放メソッド３１ｂは、リソース１４の解放を行ない、解放したリソース１４の個数を１つ増やす操作を行なう関数であり、調停メソッド３１ｃは、リソース１４の種類（“Ｒ１”，“Ｒ２”）毎にそれぞれの「調停ルール」に従っ

てリソース割り当ての調停を行なうための関数で、破線枠３１１内の記述により、リソース要求キュー２０が空でない限り、リソース要求キュー２０から１つずつ要求１９が取り出され、例えば、リソース１４の種類が“Ｒ１”の場合は破線枠３１２及び３１３内の記述が実行され、リソース１４の種類が“Ｒ２”の場合は破線枠３１４及び３１５内の記述が実行される。

【００４２】

なお、破線枠３１２，３１４内の記述は、要求されたリソース１４（“Ｒ１”もしくは“Ｒ２”）の個数がその時点で０以下であれば、割り当てられるリソース１４が無いので、リソース割り当て結果の回答２１として“False”をリソース回答キュー２２に格納することを意味する。

また、破線枠３１３，３１５内の記述は、上記以外の場合、即ち、要求されたリソース１４（“Ｒ１”もしくは“Ｒ２”）の個数が１以上であれば、割り当て可能なので、そのリソース１４の調停ルールに従ってリソース１４の割り当てを行なって、その調停（割り当て）結果（“True”）を回答２１としてリソース回答キュー２２に格納することを意味する。

【００４３】

以上のような構成により、本実施形態のシミュレーションプログラム１０は、計算機１（ＣＰＵ４）を、次のように動作（機能）させることが可能である。

即ち、図３に模式的に示すように、まず、テストベンチ１５が規定の「テスト生成ルール」に従ってテストデータ１６を生成して（ステップＳ１）、入力キュー１７に追加する（ステップＳ２）。

【００４４】

テストベンチ１５は、全てのテストデータ１６を生成した後、スレッドマネージャ１１に実行権利を譲る（ステップＳ３）。実行権利を譲り受けたスレッドマネージャ１１は、入力キュー１７からテストデータ１６を取り出して（ステップＳ４）、それに対応するスレッド１３を生成し（ステップＳ５；スレッド生成ステップ）、実行待ちキュー１８に追加する（ステップＳ６）。なお、この時点でのスレッド１３は何も実行していない未開始状態にある。

【００４５】

そして、スレッドマネージャ１１は、入力キュー１７が空になってから実行待ちキュー１８から実行待ちのスレッド１３を取り出し、そのスレッド１３が未開始状態であれば実行させて「実行状態」にする（ステップＳ７）。「実行状態」となったスレッド１３は、リソースマネージャ１２に対して最初の「メソッド」に必要なリソース１４の確保を要求する（例えば、リソース「Ａ」～「Ｃ」のうちのリソース「Ｂ」に対するリソース要求１９を発行する；ステップＳ８；リソース要求ステップ）。このリソース要求１９は、スレッドマネージャ１１によって、リソース要求キュー２０に追加される。また、このリソース要求１９を発行したスレッド１３は、実行待ちキュー１８の最後に追加される。

【００４６】

その後、スレッドマネージャ１１は、実行待ちキュー１８に格納されている全てのスレッド１３の処理（実行）が終了すると、実行権利をリソースマネージャ１２に譲る（ステップＳ９）。実行権利を譲り受けたリソースマネージャ１２は、リソース要求キュー２０の先頭から、順次、リソース要求１９を取り出して（ステップＳ１０）、現在の「調停ルール」に従ってリソース１４の割り当て調停を行ないながら、要求されたリソース１４のスレッド１３への割り当てを行ない、その結果（割り当てられたか否か；“True” or “False”）を回答２１として発行し、リソース回答キュー２２に追加してゆく（ステップＳ１１；リソース割り当てステップ）。

【００４７】

そして、リソース要求キュー２０に格納されている全ての要求１９に対するリソース割り当て処理が終了してから、リソースマネージャ１２は、実行権利をスレッドマネージャ１１に譲る（ステップＳ１２）。実行権利を譲り受けたスレッドマネージャ１１は、リソース回答キュー２２から回答２１を、順次、取り出して（ステップＳ１３）、その回答２１の内容が“True”（割り当てＯＫ）であれば、スレッドＩＤに対応するスレッド１３（この時点で、実行待ちキュー１８の先頭に位置している）を実行させて（スレッド制御ステップ）、次の「メソッド」の実行に必要なリソース要求１９を発行し、これをリソース要求キュー２０に追加してから、実行したスレッド１３を再び実行待ちキュー１８の最後に追加す

る。

【００４８】

なお、リソース回答キュー２２から取り出した回答２１が“False”（割り当てＮＧ）であった場合、スレッドマネージャ１１は、現在の（前回発行した）リソース要求１９を再びリソース要求キュー２０に追加してから、スレッド１３を、「待ち状態」として再度、実行待ちキュー１８の最後に追加する（スレッド制御ステップ）。また、全ての処理を終えた（リソース１４の割り当てが完了した）スレッド１３は消滅させて、実行待ちキュー１８からそのスレッド１３を削除する。

【００４９】

以上のようにして、スレッドマネージャ１１とリソースマネージャ１２とが連携して、全てのスレッド１３の実行（全ての「メソッド」の実行）が完了するまでリソース要求，リソース割り当て及び回答２１に応じたスレッド１３の実行状態の制御を繰り返し実行することによって、論理装置の動作シミュレーションを実施する。

【００５０】

つまり、本実施形態のスレッドマネージャ１１及びリソースマネージャ１２は、それぞれ、図４に模式的に示すように、以下の各機能部を有しており、スレッド１３の実行が完了するまで、これらのスレッドマネージャ１１とリソースマネージャ１２とが連携して上記のリソース要求１９及びスレッド１３の実行状態の制御を繰り返し実行して、その実行結果を例えばディスプレイ３に出力することによって、論理装置の動作完了までの動作をシミュレーションするようになっているのである。

【００５１】

スレッドマネージャ１１

・テストデータ１６の入力により、論理装置の設計仕様に応じてその論理装置の動作完了までに必要となる機能（処理）を表したスレッドを生成するスレッド生成手段１１０としての機能

・スレッド１３の実行に必要なリソース１４をリソースマネージャ１２に要求

するリソース要求手段１１１としての機能

・このリソース要求手段１１１による要求１９に対するリソースマネージャ１２による割り当て結果（回答２１）に応じてスレッド１３の実行状態を制御するスレッド制御手段１１２としての機能

リソースマネージャ１２

・リソース要求１９に応じたリソース１４を予め規定された「調停ルール」に従ってスレッド１３に割り当てるリソース割り当て手段１２１としての機能

【００５２】

換言すれば、本実施形態では、論理装置に実装すべき「処理（機能）」が必要とするハードウェア資源に関する記述はスレッド１３（「メソッド」）には行なわず、そのスレッド１３の「メソッド」が実行されるときに、その都度、必要なリソース１４がリソースマネージャ１２によって動的に割り当てられるようになっているのである。つまり、従来のように実現すべき「機能」にアーキテクチャは反映されていないのである。

【００５３】

従って、論理装置の設計初期段階における動作シミュレーションが実現できて設計初期段階において、仕様によって要求される「機能」を正確に実現できているかといった機能（アルゴリズム）の検証が行なえるとともに、アーキテクチャに変更が必要になった場合でも、「機能」の記述変更は殆ど行なわずに、リソース１４やリソース１４の「調停ルール」，スレッド１３のスケジューリング，リソース１４に対する要求などを、その変更に応じて変更するだけで、簡単に設計の再検証を行なうことが可能になる。

【００５４】

なお、このようにリソース１４やリソースの「調停ルール」，スレッドのスケジューリング，リソースに対する要求などを変更するだけで設計の再検証を行なえるということは、「機能」に適したアーキテクチャの探索や、「機能」をアーキテクチャへマッピングする際のアーキテクチャの性能評価，設計初期段階におけるタイミングの検証なども行なえることを意味する。

【００５５】

（Ａ１）第１実施形態の具体例の説明

さて次に、以下では、上述したシミュレーション方法について、具体例を挙げて、より詳細に説明する。即ち、ここでは、ＱｏＳ（Quality of Service）システムの設計とその動作シミュレーションを行なう場合について説明する。なお、ここでいう「ＱｏＳシステム」とは、ＩＰ（Internet Protocol）ルータなどのパケット転送装置において、例えば図７に模式的に示すように、入力（受信）パケット４１を或るルールに従ってクラス分けして複数のパケットキュー４５に格納し、最終的に、これらの各キュー４５から規定のサービスレートを満足するよう〔規定のサービスレートに応じた重み付け（Ｗ０～Ｗ３など）に応じて〕パケット４１を出力するというものである。「ＱｏＳ」の詳細については例えば、文献「インターネットＱｏＳ」（Paul Ferguson, Geoff Huston 共著，戸田 巌 監訳，(株)オーム社，2000.）に記載されている。

【００５６】

①仕様からの機能の抽出

まず、最初のステップとして、図８に示すように、仕様４０から実現すべきＱｏＳシステムの「機能」を抽出する必要がある（ステップＳ２０）。即ち、この場合は、以下の（ａ）～（ｃ）に示す機能が最低限必要になる。

【００５７】

（ａ）パケット４１のクラス分け機能４２（図７参照）：Classify( )

この機能４２は、パケットヘッダの「IP Precedenceフィールド」の値によってクラス分けする機能で、本実施形態では、例えば次表１に示すルールに従ってクラス分けを行なうものと仮定する。

【００５８】

【表１】

クラス分け

IP Precedence	クラス	対応するキュー番号
000	0	0
001		
010	1	1
011		
100	2	2
101		
110	3	3
111		

【００５９】

（ｂ）パケットをキュー４５に格納する（キューイング）機能４３：Queuing( )

この機能４３は、上記のクラス分け機能４２によってクラス分けされたパケット４１を対応するキュー４５に格納する機能である。ここで、各キュー４５には格納できる最大パケット数（以下、最大サイズという）が予め決められており、最大サイズを超える分のパケット４１については廃棄されるものとする。

【００６０】

（ｃ）キュー４５からパケットを取り出す（デキュー）機能４４：Dequeuing( )

このデキュー機能４４は、各キュー４５からパケット４１を取り出す（出力する）機能であるが、各キュー４５には予め「サービスレート」が定められており、この「サービスレート」によってキュー４５からパケット４１を出力するか否かを決定する。

【００６１】

②「機能」のスレッド化

次に、上記の各機能４２～４４を「メソッド」としてスレッド化する（図８の

ステップＳ２１）。即ち、前記の処理Ａ→処理Ｂ→処理Ｃという一連の処理（機能）を、図５により前述したように次のような１つのスレッド１３として表す。

Classify( ) → Queuing( ) → Dequeuing( )

ここで、このスレッド１３の状態は次表２に示すように３種類とする。

【００６２】

【表２】

スレッド状態

スレッド状態	説明
実行状態	スレッドが実行されている状態
待ち状態	スレッドが事象を待っている状態
終了状態	スレッドが終了されている状態

【００６３】

なお、上記の各状態はリソース１４の確保状況（割り当て結果），キューイング機能４２〔Queuing( )〕，デキュー機能４３〔Dequeuing( )〕の結果に依存する。例えば、Queuing( )を実行しようとしたときに、キュー４５が既にいっぱいになっていて、パケット４１を廃棄しなければならない場合には、スレッド１３は「実行状態」から「終了状態」に変わる。また、リソース１４が確保できない場合は、そのスレッド１３は図３により上述したように実行待ちキュー１８の最後に追加されて「待ち状態」となる。

【００６４】

③仕様４０からのアーキテクチャの設計

さて、上記の一方で、仕様４０を基にＱｏＳシステムのアーキテクチャを設計する必要がある（ステップＳ２２）。即ち、例えば、上記のClassify( )とQueuing( )とを或るリソース１４（仮に、Ｒ１とする）の上に実現し、Dequeuing( )を別のリソース１４（仮に、Ｒ２とする）の上に実現する。

【００６５】

ここで、上記のClassify( )がリソースＲ１上に実現されているとき〔つまり

、Classify( )がリソースＲ１を確保して実行中〕の実行遅延時間をTlc、同様に、Queuing( )がリソースＲ１上で実現されているときの実行遅延時間をT_{1q}、Dequeuing( )がリソースＲ２上で実現されているときの実行遅延時間をT_{2d}とする。

【００６６】

また、リソースＲ１の競合が発生した場合は、Classify( )からの要求を優先し、リソースＲ２の競合が発生した場合は、要求の到着順位でリソースを割り当てるものとする。さらに、上記Dequeuing( )の性能をアップするため、リソースＲ２の数を例えば２つとする。

【００６７】

④アーキテクチャからのリソースの抽出

以上により、リソース１４（Ｒ１，Ｒ２）の個数及び調停ルールが次表３に示すように定義される（図８のステップＳ２３）。

【００６８】

【表３】

リソース一覧

リソース名	個数	調停ルール
Ｒ１	1	Classify()が優先される。
Ｒ２	2	先着要求が優先される。

【００６９】

④スレッド１３へのリソース要求の取り込み

次に、上記のClassify( )，Queuing( )およびDequeuing( )を実行するために、上記のリソースＲ１とＲ２を確保しなければならない（スレッド１３にリソース要求を組み込む；図８のステップＳ２４）。即ち、以下の手順（１）～（５）を実現（記述）する必要がある。

【００７０】

（１）パケット４１が到着したら、スレッド１３が動的に生成され、スレッド

１３が実行状態になる。

（２）Classify( )を実行するために、リソースＲ１を確保しようとする（リソースＲ１についての要求１９を発行する）。確保できたら、次へ進む〔Classify( )を実行する〕。確保できなかったら、そのスレッド１３は「待ち状態」になる。次のステップで（次に、スレッド１３が「実行状態」となったときに）、再度、リソースＲ１の確保を行なう。

【００７１】

（３）Classify( )を実行した後、Queuing( )を実行するために、リソースＲ１を確保しようとする。確保できたら、次へ進む〔Queuing( )を実行する〕。確保できなかったら、スレッド１３は「待ち状態」となり、次のステップで（次にスレッド１３が「実行状態」となったときに）、再度、リソースＲ１の確保を行なう。

【００７２】

（４）Queuing( )を実行した後、Dequeuing( )を実行するために、リソースＲ２を確保しようとする。確保できたら、次へ進む〔Dequeuing( )を実行する〕。確保できなかったら、スレッド１３は「待ち状態」となり、次のステップで（次にスレッド１３が「実行状態」となったときに）、再度、リソースＲ２の確保を行なう。

（５）全ての処理が終了したので、スレッド１３を終了させる。

【００７３】

⑥リソース調停ルールの記述

次に、複数のスレッド１３が同じリソース１４（Ｒ１又はＲ２）に対して要求した場合、リソース１４の競合が起きるため、「調停ルール」を定義する必要がある（図８のステップＳ２５）。

【００７４】

まず、リソースＲ１の「調停ルール」（図６に示した破線枠３１３内における「Ｒ１の調停ルール」に相当）は以下の手順が実現されるように記述する。

（１）現在、リソースＲ１が空いているかどうかを判別する。空いていなければ、全ての要求１９に対して、割り当てられない旨の回答２１を発行する。空い

ていれば、次へ進む。

【００７５】

（２）単位時間内の要求１９の中にClassify( )を実行するためのリソース要求があれば、そのスレッド１３（複数ある場合は最先に要求１９を出したスレッド１３）にリソースＲ１を割り当てる。Classify( )を実行するためのリソース要求１９がなければ、最先に要求１９を出したスレッド１３にリソースＲ１を割り当てる。それ以外の要求１９に対しては割り当てられない旨の回答２１を発行する。

【００７６】

一方、リソースＲ２の「調停ルール」（図６に示した破線枠３１４内における「Ｒ２の調停ルール」に相当）は以下の手順が実現されるように記述する。

（１）現在、リソースＲ２が空いているかどうかを判別する。空いていなければ、全ての要求１９に対して、割り当てられない旨の回答２１を発行する。空いていれば、次へ進む。

【００７７】

（２）単位時間内の要求１９の中で最先に要求を出したスレッド１３にリソースＲ１を割り当てる。それ以外の要求１９に対しては割り当てられない旨の回答２１を発行する。

【００７８】

⑦シミュレーション

以上のようにしてＱｏＳシステムに必要な「機能」をプログラミングし（シミュレーションプログラム１０を構築し）、計算機１（ＣＰＵ４）上で実行させることで、ＱｏＳシステムの動作シミュレーションを実施する（図８のステップＳ２６）。即ち、計算機１（ＣＰＵ４）が、図３及び図４により前述したように、スレッドマネージャ１１及びリソースマネージャ１２として動作することにより、ＱｏＳシステムの動作シミュレーションが実施される。

【００７９】

⑧シミュレーション結果と予測値の比較

上記のシミュレーションにより或るシミュレーション結果が得られる。即ち、

例えば、パケット４１が或るレートで出力されることが分かる。そして、このシミュレーション結果と、仕様４０から予測される動作結果（予測値）４０′とを比較する（図８のステップＳ２７；比較ステップ）ことにより、設計したアルゴリズムが正しい（ステップＳ２７のＹＥＳルートからステップＳ２８）か否（設計ミス）か（ステップＳ２７のＮＯルートからステップＳ２９）が判断され、その結果が、例えば外部装置としてのディスプレイ３などに出力（表示）される（出力ステップ）。

【００８０】

つまり、図８に破線枠５１内に示す部分は、上述した論理装置のシミュレーション方法によるシミュレーション結果と、その論理装置の動作予測値とを比較する比較ステップ（Ｓ２７）と、この比較ステップ（Ｓ２７）での比較結果をディスプレイ３などの外部装置へ出力する出力ステップとを、計算機１（ＣＰＵ４）に実行させて、論理装置の機能検証を行なうための（動作シミュレーション）プログラムとして機能する。

【００８１】

なお、本プログラム５１も、例えばＦＤ９やＣＤ－ＲＯＭ，ＭＯなどの記録媒体に記録することができ、その記録媒体に記録されたプログラム５１を計算機１にインストールする（ハードディスク６に記憶する）ことで、計算機１（ＣＰＵ４）を上述のごとく動作させることが可能になる。

【００８２】

そして、設計ミス（例えば、廃棄されてしまうパケット４１の数が規定よりも多いなど）であれば、キュー４５のサイズを変える〔キュー４５（リソース１４）に関する記述を変更する〕などして、最適なアーキテクチャが得られるまで、上記のシミュレーションを行なって、ＱｏＳシステムに最適なアーキテクチャを探索する。

【００８３】

なお、上記の予測値は、上記の例の場合、前述したClassify( )の実行遅延時間Tlc，Queuing( )の実行遅延時間T_{1q}およびDequeuing( )の実行遅延時間T_{2d}を基に算出できる。また、上記のシミュレーション結果や予測値との比較結果は、

ディスプレイ３だけでなく、プリンタなどの周辺機器に出力することも可能である。

【００８４】

以上のようにして、論理装置としてのＱｏＳシステムの機能検証（システムとしての要求を満足しているか否かの検証）を、システムの設計初期段階で実施することができる。また、アーキテクチャに変更が必要になった場合でも、使用するリソース１４（Ｒ１，Ｒ２）やリソース１４の「調停ルール」，スレッド１３のスケジューリング，リソース１４に対する要求などを、その変更に応じて変更するだけで、簡単に設計（アルゴリズム）の再検証を行なうことができる。従って、手戻り、記述スタイルの変更が少ない柔軟性の高い設計手法（つまり、再利用性に優れた記述スタイル）を実現することが可能となり、システムの設計コストを大幅に削減することができる。

【００８５】

（Ｂ）第１変形例の説明

ところで、上述したリソースマネージャ１２には、例えば図９に示すように、デッドロック検出機構１２２を追加してもよい。このデッドロック検出機構１２２は、複数のスレッド１３がリソース１４を取り合うことによって「デッドロック」に陥る可能性があるか否かを検出するためのものである。

【００８６】

なお、ここでいう「デッドロック」とは、２つ以上のスレッド１３が互いに相手が占有しているリソース１３を要求して解決しない特定の状態になっていることを意味する。「デッドロック」については、例えば、文献「現代オペレーティングシステムの基礎」（萩原宏，津田孝夫，大久保英嗣共著，（株）オーム社，pp.63，1995.）で説明されている。

【００８７】

例えば、それぞれ図１０に示すように記述されたスレッド１３が２つ（仮に、スレッド“１”，“２”とする）存在していた場合を考える。ただし、ここでのリソース１４（“Ａ”）の数は３個、リソース１４（“Ｂ”）の数は２個と仮定する。また、この図１０において、「リソースＡ．要求（ ）」などの関数（括

弧）内に表示された数値（この場合は“２”）はスレッドＩＤを表し、以降の説明においても同様とする。

【００８８】

そして、このような場合、各スレッド“１”，“２”のそれぞれのリソース割付けグラフ（resource allocation graph）は、図１１（Ａ）に示すようになる。

この図１１（Ａ）に示す状態を簡約（reduce）して表すと、図１１（Ｂ）に示すような状態となるが、この状態で既にそれ以上の簡約が不可能（irreducible）な状態であることが分かる。

【００８９】

これは、スレッド“１”とスレッド“２”との間で「デッドロック」が発生する可能性があることを意味する。デッドロック検出機構１２２は、このようにリソース割付けグラフを基に「デッドロック」の発生可能性を判断する。

こうすることで、設計の初期段階において「デッドロック」の発生可能性を検出することが可能となり、論理装置の機能検証を行なうことが可能となる。

【００９０】

（Ｃ）第２変形例の説明

さて、上記のリソースマネージャ１２には、例えば図１２に示すように、リード／ライト検出機構１２３を追加してもよい。ここで、このリード／ライト検出機構１２３は、次のような事象（１）～（３）を監視（検出）することにより、リード／ライトエラーの発生可能性を検出するためのものである。

【００９１】

（１）同一リソース１４に対するリード要求とライト要求の同一単位時間内の発生

（２）既にライト要求によって占有されているリソース１４に対するリード要求の発生

（３）既にリード要求によって占有されているリソース１４に対するライト要求の発生

【００９２】

つまり、このリード／ライト検出機構１２３の検出結果をみれば、複数のスレッド１３がリソース１４に対してシーケンシャルに書き込み動作もしくは読み出し動作を行なう時に、その順序が正しいか否かを判定（つまり、複数スレッド１３間のリソース１４に対するリード／ライト動作の競合状態を判定）して、設計した論理装置が正しく動作するか否かを検証することができるのである。

【００９３】

これを実現するには、要求１９の記述を例えば図１３に示すように拡張する。即ち、次表４に示すような意味をもつ「リード／ライトフラグ」を定義する。

【００９４】

【表４】

リード／ライトフラグ＝０	リード／ライト要求を考慮しない
リード／ライトフラグ＝１	リード要求
リード／ライトフラグ＝２	ライト要求

【００９５】

加えて、リソース１４の記述を例えば図１４に示すように拡張する。即ち、リソース１４の解放が行なわれた場合には、必ず「カレントフラグ（CurrentFlag）」を“０”にするようにしておき（破線枠３１６参照）、この「カレントフラグ」が“０”でないときに、要求１９の「リード／ライトフラグ」とリソース１４側の「カレントフラグ」とが一致しなければ、「リード／ライトエラーの発生可能性がある」といったエラー情報がディスプレイ３などに出力される（破線枠３１７参照）ようにするのである。

【００９６】

（Ｄ）第３変形例の説明

さらに、上記のリソースマネージャ１２には、例えば図１５に示すように、ボトルネック検出機構１２４を追加してもよい。ここで、このボトルネック検出機構１２４は、スレッド１３の実行結果やリソース１４に対するアクセス状況を基

に「ボトルネック」の検出を行なうためのもので、例えば、リソース１４の個数と種類を制限した環境下でシミュレーションを行なって、各リソース１４に対するアクセス回数やリソース１４を割り当てられなかった時の待ち時間などを確認することによって、論理装置の「ボトルネック」となっている部分を検出することが可能となる。

【００９７】

具体的には、例えば図１６に示すように、リソース１４の記述に、要求の数（アクセス数）を計数する「メソッド」を加えるとともに、その計数結果を「要求合計メンバ関数〔要求合計( )〕」によって呼び出せるようにしておけば、リソース１４のアクセス回数を監視する仕組みがリソースマネージャ１２において実現される。

【００９８】

つまり、ボトルネック検出機構１２４によって、各リソース１４の「要求合計メンバ関数」を呼び出せば、それまでの各リソース１４に対するアクセス頻度を調べることができ、例えば、アクセス頻度が最も多いリソース１４は論理装置の「ボトルネック」になっている可能性があると推測することができる。従って、論理装置の設計初期段階で論理装置の性能検証（ボトルネックの有無の検証）を行なうことができる。

【００９９】

（Ｅ）第４変形例の説明

また、上記のリソースマネージャ１２には、例えば図１７に示すように、ブロッキング検出機構１２５を追加してもよい。ここで、このブロッキング検出機構１２５は、スレッド１３からのリソース１４の要求１９が所定の制限数を超えた場合の「ブロッキング」の発生状況を調べる機能を提供するもので、リソース１４に例えば図１８に示すような記述を加えることで実現される。

【０１００】

そして、リソース１４の「個数」を制限した（予め定めておく）環境下でシミュレーションを行なうことで、割り当てられるリソース数を超える要求１９があった場合〔if文の条件（「個数」＜０）を満足する場合〕には「ブロッキング」

が発生したと考えられ、例えば「リソースＡの要求はブロッキングする必要がある。」といったエラー情報をディスプレイ３などに出力する。

【０１０１】

このようにして、本変形例では、ブロッキング検出機構１２５により、リソース１４の「個数」を制限した環境下でのスレッド１３からのリソース要求が制限数を超えた場合のブロッキングの発生可能性を調べることで、論理装置の性能を評価することができる。

【０１０２】

なお、上述した第１～第４変形例におけるデッドロック検出機構１２２，リード／ライト検出機構１２３，ボトルネック検出機構１２４，ブロッキング検出機構１２５は、論理装置の検証項目に応じて、全てあるいは一部の組み合わせでそなえられていてもよい。

【０１０３】

（Ｆ）第５変形例の説明

ところで、上記のスレッド１３には、そのスレッド１３に記述されている一連の処理（「機能」；「メソッド」）に必要な実行時間の見積もり値（つまり、リソースマネージャ１２によって割り当てられたリソース１４を占有する時間についての予算）である「タイムバジェット」を各「機能」に対してそれぞれ設定してもよい。

【０１０４】

即ち、図８により前述したようにアーキテクチャからリソース１４を抽出して各リソース１４に処理をマッピングする際に（ステップＳ２４にてスレッド１３にリソース要求を組み込んだ後）、各処理の見積もり時間（タイムバジェット）をスレッド１３に追加する（図１９のステップＳ２５′）。

【０１０５】

例えば図２０に示すように、スレッド１３に記述される「機能」を処理１，処理２とすると、それぞれに対してdelay(処理１のタイムバジェット)，delay(処理２のタイムバジェット)を設定するのである。

そして、仕様４０から論理装置がｎ個の入力データを処理する際の実行時間の

制限をＴとし（図１９のステップＳ２６ｂ）、テストベンチ１５からｎ個のデータを生成させて、前述したごとくシミュレーションを実行する（ステップ２６ａ）。

【０１０６】

その後、全てのスレッド１３が消滅されるまでの時間ｔを計測し、その計測時間ｔが制限時間Ｔ内であれば正しい設計と判断できる（図１９のステップＳ２７′のＹＥＳルートからステップＳ２８′）。逆に、計測時間ｔが制限時間Ｔよりも大きい場合には仕様４０の性能要件を満たしていないと判断できる（図１９のステップＳ２７′のＮＯルートからステップＳ２９′）。

【０１０７】

このようにして、各種「機能」の組み合わせで構成される論理装置が、仕様４０により要求される性能要件を満たしているか否かを、設計の初期段階にて確認することができる。

【０１０８】

（Ｇ）第６変形例の説明

なお、上記のスレッド１３には、例えば図２１に示すように、上記のステップＳ２４にてスレッド１３にリソース要求を組み込む前に、図２２に示すようにして「ライフタイム」を設定してもよい（ステップＳ２１′）。なお、ここでいう「ライフタイム」とは、「機能」の実行制限時間を表す。

【０１０９】

そして、第５変形例にて上述したようにスレッド１３に「タイムバジェット」を設定（ステップＳ２５′）した上で、シミュレーションを実行し（ステップＳ２６″）、スレッド１３が消滅する前にJudgeLifeTime( )関数（図２２参照）を（例えばリソースマネージャ１２から）呼び出せば、上述した「タイムバジェット」による論理装置の性能検証に加えて、そのスレッド１３に設定した「ライフタイム」が切れたか否か、つまり、「ライフタイム」内に処理が完了したか否かを判定することができる（ステップＳ２７″）。

【０１１０】

この判定の結果、「ライフタイム」内にそのスレッド１３の処理が完了してい

れば、論理装置が仕様４０により要求されるリアルタイム性に関する性能要件を満足している（正しい設計である）と判断でき（ステップＳ２７″のＹＥＳルートからステップＳ２８″）、逆に、「ライフタイム」内に処理が完了していなければ、リアルタイム性に関する性能要件を満足していないと判断できる（ステップＳ２９″）。

【０１１１】

このように、本変形例では、スレッド１３に「タイムバジェット」と「ライフタイム」とを設定することで、各種「機能」の組み合わせで成る論理装置全体としての処理時間に関する性能要件とリアルタイム性に関する性能要件とを、論理装置の設計初期段階において検証することが可能となる。

なお、本変形例では、「タイムバジェット」と「ライフタイム」の双方を設定しているが、勿論、「ライフタイム」のみを設定して、論理装置のリアルタイム性のみを検証するようにしてもよい。

【０１１２】

また、これらの「タイムバジェット」，「ライフタイム」は、上述した例ではスレッド１３に記述しているが、リソース１４に記述することも可能である。例えば、メモリやバッファなどの記憶素子でのデータ保持時間が規定されるケースであれば、その記憶素子（リソース）に対して「タイムバジェット」や「ライフタイム」もしくはその両方を記述してもよい。

【０１１３】

（Ｈ）第７変形例の説明

上述した実施形態では、論理装置の動作完了までに必要な一連の処理（機能）を１つのスレッド１３に表した場合について説明したが、上記「一連の処理」は、例えば図２３に模式的に示すように、複数の逐次的なスレッド１３（スレッド“１”～スレッド“ｎ”）に表されていてもよい。

【０１１４】

この場合の動作は次のようになる。即ち、外部入力（テストデータ１６の入力）によりスレッドマネージャ１１が最初のスレッド“１”を生成する。すると、スレッド“１”は処理“１”を実行して、処理“１”に必要なリソース１４をリ

ソースマネージャ１２に要求する。

【０１１５】

これにより、スレッド“１”にリソース１４が割り当てられてスレッド“１”が処理“１”を完了し、スレッドマネージャ１１によって消滅させられる際に、次のスレッド“２”を生成する。なお、リソース１４が割り当てられなかった場合は、この場合も、そのスレッド１３はスレッドマネージャ１１によって「待ち状態」に制御される。

【０１１６】

以降、スレッド“ｉ”（ただし、ｉ＝１～ｎ－１）が消滅する際にそのスレッド“ｉ”が次のスレッド“ｉ＋１”を生成する。このようにして、論理装置の入力から動作完了までの一連の処理を一連の逐次的なスレッド１３に表現する。つまり、本変形例は、スレッドマネージャ１１が行なうスレッド１３の生成を、スレッド１３からも生成できるようにしたものである。

【０１１７】

具体的に、これを実現するには、例えば図２４に示すように、スレッド１３に対して「終了を待つ( )」と言う「メソッド」を追加するとともに、処理“１”，処理“２”，処理“３”，…に対応したスレッド“１”，スレッド“２”，スレッド“３”，…を追加する。

【０１１８】

これにより、各処理“１”，“２”，“３”，…に対して１つずつスレッド１３が生成され、それぞれのスレッド１３の生成タイミングは前のスレッド１３が処理を終了した後になる。これにより、スレッド１３間の依存関係が前述した実施形態よりもさらに明確になる。

【０１１９】

そして、このように構成したスレッド１３を図３により前述したシステムに組み込んで設計の初期段階における論理装置のシミュレーションを行なう。即ち、この場合も、全てのスレッド１３の処理が完了（全てのスレッド１３が消滅）するまで、スレッドマネージャ１１とリソースマネージャ１２とが連携して、リソース１４の要求，リソース１４の割り当て及びリソース１４の割り当て結果に応

じたスレッド１３の実行状態の制御を繰り返すことによって、論理装置の動作シミュレーションを実現する。

【０１２０】

このように、本変形例では、論理装置の入力から動作完了までの一連の処理を一連の逐次的なスレッド１３に表現して、シミュレーションを行なうので、論理装置に必要な処理の依存関係を明確にした上で、その機能検証を設計初期段階において行なうことができる。

【０１２１】

（Ｉ）第８変形例の説明

なお、上記の論理装置の動作完了までの一連の処理は、例えば図２５に模式的に示すように、複数の逐次的または並行に実行されるスレッド１３で表してもよい。即ち、第７変形例にて上述したようにスレッド１３が消滅する際に、そのスレッド１３が複数のスレッド１３を同時に生成して各スレッド１３を並列に実行させ、これら複数のスレッド１３が消滅する際に、さらに次のスレッド１３を逐次的に生成するようにしてもよい。

【０１２２】

この場合の動作は次のようになる。即ち、外部入力（テストデータ１６の入力）によりスレッドマネージャ１１が最初のスレッド“１”を生成する。スレッド“１”は処理“１”を実行して、処理“１”に必要なリソース１４をリソースマネージャ１２に要求する。これにより、スレッド“１”にリソース１４が割り当てられてスレッド“１”が処理“１”を完了すると、スレッドマネージャ１１によって消滅させられる際に、次のスレッド“２”～“ｎ”を同時に生成する。なお、リソース１４が割り当てられなかった場合は、この場合も、そのスレッド１３はスレッドマネージャ１１によって「待ち状態」に制御される。

【０１２３】

その後、各スレッド“２”～“ｎ”が並行して処理を進めてそれぞれの処理が完了し、これらの各“２”～“ｎ”がスレッドマネージャ１１によって消滅させられる際に、次のスレッド“ｎ＋１”を生成する。以降は、上述した第７変形例と同様にして、順次、前のスレッド１３が消滅する際に、次のスレッド１３が生

成されて処理が実行される。

【０１２４】

例えば、処理Ａと処理Ｂの実行結果を処理Ｃが使う場合、スレッド１３の構成例は図２６に示すようになる。そして、このように構成したスレッド１３を、この場合も、図３により前述したシステムに適用して設計の初期段階における論理装置のシミュレーションを行なう。

【０１２５】

即ち、全てのスレッド１３の処理が完了（全てのスレッド１３が消滅）するまで、スレッドマネージャ１１とリソースマネージャ１２とが連携して、リソース１４の要求，リソース１４の割り当て及びリソース１４の割り当て結果に応じたスレッド１３の実行状態の制御を繰り返すことによって、論理装置の動作シミュレーションを実現する。

【０１２６】

このように、本変形例では、スレッド１３に並列性を導入して処理の並列化を図ることにより、上述した実施形態や第７変形例よりも複雑な「機能」をもった論理装置の動作シミュレーションを実現することができる。

【０１２７】

（Ｊ）第９変形例の説明

次に、上述したリソースマネージャ１２は、リソース１４の種類に関係無く全てのリソース１４を共通で管理していたが、例えば図２７に模式的に示すように、複数種類のリソース１４－１～１４－ｎ毎にリソースマネージャ１２－１～１２－ｎを設けて、それぞれが１種類のリソース１４－ｊ（ｊ＝１～ｎ）を独立して（ローカルに）管理して、要求１９に応じたリソース１４－ｊをローカルな「調停ルール」に従って要求元のスレッド１３に割り当てるようにしてもよい。

【０１２８】

この場合は計算機１（ＣＰＵ４）を次のように動作させることが可能になる。即ち、外部入力（テストデータ１６の入力；ステップＳ３１）によりスレッドマネージャ１１がスレッド１３を生成する（ステップＳ３２）。この場合も、スレッド１３は独立して進行するが個々の実行順序はスレッドマネージャ１１に管理

されている。そして、スレッド１３は処理を進めるためにリソース１４－ｊを確保する必要がある時は、そのリソース１４－ｊの確保をスレッドマネージャ１１経由で対応するリソースマネージャ１２－ｊに要求する（ステップＳ３３；リソース要求ステップ）。

【０１２９】

リソースマネージャ１２－ｊは単位時間内のリソース要求１９をリソース１４－ｊ毎にローカルに調停して、リソース１４－ｊの割り当て結果をスレッドマネージャ１１に回答２１を発行する（ステップＳ３４；リソース割り当てステップ）。スレッドマネージャ１１はリソースマネージャ１２－ｊからの回答２１に基づいてスレッド１３の実行状態を制御する（スレッド制御ステップ）。

【０１３０】

そして、この場合も、全てのスレッド１３の処理が完了するまで、図３により前述したごとく、スレッドマネージャ１１とリソースマネージャ１２－ｊとが連携して、リソース要求，リソース割り当て及びスレッド１３の制御とを繰り返すことによって、実行結果を出力して（ステップＳ３５）、論理装置の動作シミュレーションを実現する。

【０１３１】

ここで、具体的に、リソース種類をＡ，Ｂ，Ｃの３種類（以下、リソースＡ，Ｂ，Ｃと表記する）とした場合、リソースマネージャ１２－ｊが管理（参照）するリソースＡは、Ｃ＋＋記述スタイルで構成（記述）すると例えば図２８に示すように構成される。即ち、前述したリソース要求キュー２０とリソース回答キュー２２とを定義するとともに、要求メソッド３１ａ′、リソースＡの解放を行なう解放メソッド３１ｂ′、リソースＡの調停を行なう調停メソッド３１ｃ′をそれぞれ定義する。

【０１３２】

ここで、要求メソッド３１ａ′は、要求したリソースＡとその要求１９を出したスレッド１３のスレッドＩＤとを要求メッセージとしてリソース要求キュー２０に登録する操作を行なうための関数であり、解放メソッド３１ｂ′は、リソースＡの解放を行ない、解放したリソースＡの個数を１つ増やす操作を行なう関数

である。

【０１３３】

また、調停メソッド３１ｃ′は、リソースＡの割り当て調停を独立して（ローカルに）行なうための関数で、破線枠３１１′内の記述により、リソース要求キュー２０が空でない限り、リソース要求キュー２０から１つずつ要求１９が取り出され、要求されたリソースＡの個数がその時点で０以下であれば破線枠３１２′内の記述が実行され、要求されたリソースＡの個数がその時点で１以上であれば破線枠３１３′内の記述が実行されるようになっている。

【０１３４】

即ち、要求されたリソースＡの個数がその時点で０以下であれば、新たに割り当てられるリソースＡが無いので、リソース割り当て結果の回答２１として“False”をリソース回答キュー２２に格納し、要求されたリソースＡの個数が１以上であれば、割り当て可能なので、そのリソースＡの調停ルールに従って割り当てを行なって、その調停（割り当て）結果（“True”）を回答２１としてリソース回答キュー２２に格納するのである。なお、リソースＢ，リソースＣについても上記（図２８）と同様に記述できる。つまり、この場合は、図６に示すものに比して記述量は増えるものの構成は簡単になる。

【０１３５】

そして、このように構成したリソースＡ，Ｂ，Ｃを図３に示すシステムに適用すれば、リソース種類毎のリソースマネージャ１２－ｊによって、互いに依存関係の無いローカルな「調停ルール」に従ったリソース割り当てが行なわれて、リソースＡ，Ｂ，Ｃの種類間に依存関係が存在しない論理装置のシミュレーションが可能となる。

【０１３６】

（Ｋ）第１０変形例の説明

なお、上述した第９変形例は、リソース種類に互いに依存関係が無い場合であるが、依存関係がある場合は、上述したローカルな調停では不十分である。例えば、「CPU」というリソースが、「論理演算装置（ALU；Arithmetic Logic Unit），「バス（BUS）」，「レジスタ（REGISTER）」などのリソースから構成され

、「ALU」というリソースが、さらに、「加算器（ADDER）」，「減算器（SUBTRACTER）」，「論理積回路（AND）」，「論理和回路（OR）」などのリソースから構成されているような場合、「CPU」というリソースをスレッド１３に割り当てるには、下位階層の「ALU」，「BUS」，「REGISTER」，「ADDER」，「AND」，「OR」といったリソースも空いていなければならない。

【０１３７】

このように、リソースに１４－ｊに依存関係がある場合には、例えば図２９に模式的に示すように、その依存関係に応じてリソースマネージャ１２－ｊを階層化する。

即ち、この図２９の場合は、リソース１４－１とリソース１４－２との間には依存関係が無いため、それぞれリソースマネージャ１２－１，１２－２がローカルにリソース１４－１，１４－２を管理するが、リソース１４－１，１４－２とリソース１４－３との間にはそれぞれ依存関係が存在するため、上位階層のリソースマネージャ１２－３が、下位階層のリソースマネージャ１２－１，１２－２も管理することで下位階層のリソース１４－１，１４－２の管理も行なえるようにするのである。

【０１３８】

同様にして、リソースマネージャ１２－３やリソースマネージャ１２－（ｎ－１）は、互いに独立してリソース１４－３や１４－（ｎ－１）の管理をローカルに行なうが、それぞれ上位階層のリソース１４－ｎと依存関係があるため、上位階層のリソースマネージャ１２－ｎが下位階層のリソースマネージャ１２－３や１２－（ｎ－１）も管理する。

【０１３９】

このようにして、リソース１４－ｊに依存関係があり前述したようなローカルなリソース調停で解決できない場合はリソース１４－ｊの階層化を行なってグローバルなリソース調停を行なう。

なお、この場合、計算機１（ＣＰＵ４）は、次のように動作することになる。即ち、外部入力（テストデータ１６の入力；ステップＳ４１）によりスレッドマネージャ１１がスレッド１３を生成する（ステップＳ４２）。この場合も、スレ

ッド１３は独立して進行するが個々の実行順序はスレッドマネージャ１１に管理されている。

【０１４０】

そして、スレッド１３は処理を進めるためにリソース１４－ｊを確保する必要がある時は、そのリソース１４－ｊの確保を、スレッドマネージャ１１経由でそのリソース種類に対応するリソースマネージャ１２－ｊに要求する（ステップＳ４３；リソース要求ステップ）。

【０１４１】

リソースマネージャ１２－ｊは、単位時間内の個々のリソース要求に対する割り当てをリソース１４－ｊ毎にローカルに調停する。ローカルなリソース１４－ｊの割り当て結果は上位階層のリソースマネージャ１２－ｋ（ｋ＝ｊ＋１）に伝わり、上位階層のリソースマネージャ１２－ｋは、他の関連する（依存関係のある）種類のリソース１２－ｋの割り当て結果を考慮してその階層のリソース割り当てを決定する。

【０１４２】

以降、同様にして、下位階層のリソース割り当て結果が、順次、上位階層に伝搬してゆき、最終的に、最上位階層のリソースマネージャ１２－ｎが最後の割り当て結果をスレッドマネージャ１１に回答する（ステップＳ４４；リソース割り当てステップ）。

【０１４３】

つまり、本変形例では、リソース割り当てステップＳ４４において、上位階層のリソースマネージャ１２－ｋが、自身の管理するリソース１４－ｋと下位階層のリソースマネージャ１２－ｊの管理するリソース１４－ｊとの依存関係を考慮してリソース割り当てを行なうのである。

【０１４４】

そして、スレッドマネージャ１１は、このリソースマネージャ１２－ｊからの回答２１に基づいてスレッド１３の実行状態を制御する（スレッド制御ステップ）。その後、全てのスレッド１３の処理が完了するまで、スレッドマネージャ１１とリソースマネージャ１２－ｊとが連携して、上記のリソース要求，リソース

割り当て及びスレッド１３の制御とを繰り返すことにより、実行結果（シミュレーション結果）を出力して（ステップＳ４５）、論理装置のシミュレーションを実現する。

【０１４５】

このように、本変形例では、リソース１４－ｊの依存関係に応じてリソース１４－ｊ及びリソースマネージャ１２－ｊを階層化することで、上位階層のリソースマネージャ１２－ｊが下位階層のリソースマネージャ１２－ｊの管理するリソース１４－ｊも考慮してリソース１４－ｊの割り当てを行なうことができるので、リソース種類間のグローバルな調停を行なうことができ、より複雑なリソースの依存関係を持つ論理装置の動作シミュレーションを実現して、設計初期段階で機能や性能検証を行なうことができる。

【０１４６】

なお、上述した第７～第１０変形例は、勿論、前述した第１～第６変形例のいずれに適用してもよく、それぞれ、組み合わせによる利点ないし効果が得られる。

以上のように、本実施形態及びその各変形例の論理装置の動作シミュレーション方法によれば、論理装置の設計初期段階において機能レベルのシミュレーションを行なって、「機能」の確認，「機能」を実現するアーキテクチャの妥当性を設計の初期段階で確認して論理装置の機能検証と性能評価を行なうことができる。

【０１４７】

また、設計の初期段階に「機能」を正確に実現できたか否かを確認することができるとともに、設計の安全性を確認することができる。その上、アーキテクチャに変更が必要となった場合でも、「機能」の記述の変更は殆ど行なわずに、リソース１４（１４－ｊ），「調停ルール」，リソース要求などを変更するだけで、簡単に設計の再検証を行なえるので、従来のように「機能」の記述の変更による設計コスト増加を回避することができる。

【０１４８】

（Ｌ）その他

上述した実施形態及びその各変形例では、シミュレーションプログラム１０の記述の具体例としてＣ＋＋記述スタイルを例にしたが、勿論、Ｊａｖａなどのなどの他のプログラミング言語に基づく記述スタイルを適用しても、上記と同様の作用効果が得られることはいうまでもない。

そして、本発明は、上述した実施形態及びその各変形例に限定されず、上記以外にも、本発明の趣旨を逸脱しない範囲で種々変形して実施することができる。

【０１４９】

（Ｍ）付記

〔付記１〕　プログラムの実行単位であるスレッドの制御を行なうスレッドマネージャが、論理装置の設計仕様に応じて該論理装置の動作完了までに必要となる機能を表したスレッドの実行に必要なハードウェアリソースを、当該ハードウェアリソースを管理するリソースマネージャに要求するリソース要求ステップと、

該リソースマネージャが、該要求に応じたハードウェアリソースを予め規定されたルールに従って該スレッドに割り当てるリソース割り当てステップと、

該スレッドマネージャが、該リソースマネージャによる割り当て結果に応じて該スレッドの実行状態を制御するスレッド制御ステップとを有するとともに、

該スレッドの実行が完了するまで該スレッドマネージャと該リソースマネージャとが連携して上記の各ステップを繰り返し実行することにより、該論理装置の動作完了までの動作をシミュレーションすることを特徴とする、論理装置の動作シミュレーション方法。

【０１５０】

〔付記２〕　上記の一連の機能が、複数の逐次的なスレッドに表されていることを特徴とする、付記１記載の論理装置の動作シミュレーション方法。

〔付記３〕　上記の一連の機能が、複数の逐次的または並行に実行されるスレッドに表されていることを特徴とする、付記１記載の論理装置の動作シミュレーション方法。

【０１５１】

〔付記４〕　該リソースマネージャが、該ハードウェアリソースの種類に対応

して複数設けられるとともに、

該リソース割り当てステップにおいて、上記の各リソースマネージャが、それぞれ、自身が管理するハードウェアリソースを予め規定されたローカルなルールに従って該スレッドに割り当てることを特徴とする、付記１～３のいずれか１項に記載の論理装置の動作シミュレーション方法。

【０１５２】

〔付記５〕　該リソースマネージャが、該ハードウェアリソースの種類に対応して複数設けられるとともに、各リソースマネージャが、該ハードウェアリソースの依存関係に応じて階層化され、

該リソース割り当てステップにおいて、該リソースマネージャが、自身の管理するハードウェアリソースと下位階層のリソースマネージャの管理するハードウェアリソースとの依存関係を考慮して該ハードウェアリソースの割り当てを行なうことを特徴とする、付記１～３のいずれか１項に記載の論理装置の動作シミュレーション方法。

【０１５３】

〔付記６〕　該リソースマネージャが、該リソース要求ステップによるリソース要求を監視し、その監視結果に基づいて複数スレッド間のリソース要求のデッドロック状態を判定することを特徴とする、付記１～５のいずれか１項に記載の論理装置の動作シミュレーション方法。

〔付記７〕　該リソースマネージャが、該リソース要求ステップによるリソース要求によって割り当てられたハードウェアリソースに対するリード／ライト要求を監視し、その監視結果に基づいて複数スレッドのハードウェアリソースに対するリード／ライト動作の競合状態を判定することを特徴とする、付記１～５のいずれか１項に記載の論理装置の動作シミュレーション方法。

【０１５４】

〔付記８〕　該リソースマネージャが、該ハードウェアリソースに対するリソース要求数を監視し、その監視結果に基づいて該スレッドのボトルネックを検出することを特徴とする、付記１～５のいずれか１項に記載の論理装置の動作シミュレーション方法。

〔付記９〕　該リソースマネージャが、該ハードウェアリソースに対するリソース要求数を監視し、その監視結果に基づいて該リソース要求のブロッキングを検出することを特徴とする、付記１～５のいずれか１項に記載の論理装置の動作シミュレーション方法。

【０１５５】

〔付記１０〕　該スレッドに、該リソースマネージャによって割り当てられたハードウェアリソースを占有する時間についての予算を与えておくことを特徴とする、付記１～５のいずれか１項に記載の論理装置の動作シミュレーション方法。

〔付記１１〕　該スレッドに、該機能の実行制限時間を与えておくことを特徴とする、付記１～５のいずれか１項に記載の論理装置の動作シミュレーション方法。

【０１５６】

〔付記１２〕　付記１～１１のいずれか１項に記載の動作シミュレーション方法によるシミュレーション結果と、該論理装置の動作予測値とを比較する比較ステップと、

該比較ステップでの比較結果を外部装置へ出力する出力ステップとを有することを特徴とする、論理装置の動作シミュレーション方法。

【０１５７】

〔付記１３〕　プログラムの実行単位であるスレッドの制御を行なうスレッドマネージャと、

該スレッドの実行に必要なハードウェアリソースを管理するリソースマネージャとをそなえるとともに、

該スレッドマネージャが、

論理装置の設計仕様に応じて該論理装置の動作完了までに必要となる機能を表したスレッドの実行に必要なハードウェアリソースを該リソースマネージャに要求するリソース要求手段と、

該リソース要求手段による該要求に対する該リソースマネージャによる割り当て結果に応じて該スレッドの実行状態を制御するスレッド制御手段とをそなえ、

且つ、

該リソースマネージャが、

該要求に応じたハードウェアリソースを予め規定されたルールに従って該スレッドに割り当てるリソース割り当て手段をそなえ、

該スレッドの実行が完了するまで該スレッドマネージャと該リソースマネージャとが連携して上記のリソース要求及びスレッドの実行状態の制御を繰り返し実行することにより、該論理装置の動作完了までの動作をシミュレーションすることを特徴とする、論理装置の動作シミュレーション装置。

【０１５８】

〔付記１４〕　論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体であって、

該コンピュータを、

プログラムの実行単位であるスレッドの制御を行なうスレッドマネージャ及び該スレッドの実行に必要なハードウェアリソースを管理するリソースマネージャとして機能させるとともに、

該スレッドマネージャが、論理装置の設計仕様に応じて該論理装置の動作完了までに必要となる機能を表したスレッドの実行に必要なハードウェアリソースを該リソースマネージャに要求するリソース要求ステップと、該リソースマネージャが、該要求に応じたハードウェアリソースを予め規定されたルールに従って該スレッドに割り当てるリソース割り当てステップと、該スレッドマネージャが、該リソースマネージャによる割り当て結果に応じて該スレッドの実行状態を制御するスレッド制御ステップとを実行するとともに、該スレッドの処理が完了するまで該スレッドマネージャと該リソースマネージャとが連携して上記の各ステップを繰り返し実行することにより、該論理装置の動作完了までの動作をシミュレーションするためのプログラムが記録されていることを特徴とする、論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【０１５９】

〔付記１５〕　上記の一連の機能が、複数の逐次的なスレッドに表されている

ことを特徴とする、付記１４記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

〔付記１６〕　上記の一連の機能が、複数の逐次的もしくは並行に実行されるスレッドに表されていることを特徴とする、付記１４記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【０１６０】

〔付記１７〕　該リソースマネージャが、該ハードウェアリソースの種類に対応して複数設けられるとともに、

該リソース割り当てステップにおいて、上記の各リソースマネージャが、それぞれ、自身が管理するハードウェアリソースを予め規定されたローカルなルールに従って該スレッドに割り当てることを特徴とする、付記１４～１６のいずれか１項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【０１６１】

〔付記１８〕　該リソースマネージャが、該ハードウェアリソースの種類に対応して複数設けられるとともに、各リソースマネージャが、該ハードウェアリソースの依存関係に応じて階層化され、

該リソース割り当てステップにおいて、該リソースマネージャが、自身の管理するハードウェアリソースと下位階層のリソースマネージャの管理するハードウェアリソースとの依存関係を考慮して該ハードウェアリソースの割り当てを行なうことを特徴とする、付記１４～１６のいずれか１項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【０１６２】

〔付記１９〕　該リソースマネージャが、該リソース要求ステップによるリソース要求を監視し、その監視結果に基づいて複数スレッド間のリソース要求のデッドロック状態を判定することを特徴とする、付記１４～１８のいずれか１項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【０１６３】

〔付記２０〕　該リソースマネージャが、該リソース要求ステップによるリソース要求によって割り当てられたハードウェアリソースに対するリード／ライト要求を監視し、その監視結果に基づいて複数スレッドのハードウェアリソースに対するリード／ライト動作の競合状態を判定することを特徴とする、付記１４～１８のいずれか１項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【０１６４】

〔付記２１〕　該リソースマネージャが、該ハードウェアリソースに対するリソース要求数を監視し、その監視結果に基づいて該スレッドのボトルネックを検出することを特徴とする、付記１４～１８のいずれか１項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【０１６５】

〔付記２２〕　該リソースマネージャが、該ハードウェアリソースに対するリソース要求数を監視し、その監視結果に基づいて該リソース要求のブロッキングを検出することを特徴とする、付記１４～１８のいずれか１項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【０１６６】

〔付記２３〕　該スレッドに、該リソースマネージャによって割り当てられたハードウェアリソースを占有する時間についての予算を与えておくことを特徴とする、付記１４～１８のいずれか１項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

〔付記２４〕　該スレッドに、該機能の実行制限時間を与えておくことを特徴とする、付記１４～１８のいずれか１項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【０１６７】

〔付記２５〕　論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体であって、

該コンピュータに、

付記１～１１のいずれか１項に記載の動作シミュレーション方法によるシミュレーション結果と該論理装置の動作予測値とを比較する比較ステップと、該比較ステップでの比較結果を外部装置へ出力する出力ステップとを実行させるための動作シミュレーションプログラムが記録されたことを特徴とする、論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【０１６８】

【発明の効果】

以上詳述したように、本発明によれば、論理装置に実装すべき「機能」が必要とするハードウェアリソース（以下、単に「リソース」という）に関する記述はスレッドには行なわず、そのスレッドが実行されるときに、その都度、必要なリソースをリソースマネージャによって動的に割り当てながら、スレッドの実行が完了するまでスレッドマネージャとリソースマネージャとが連携してスレッドの実行状態の制御を繰り返し実行することにより、論理装置の動作完了までの動作をシミュレーションするので、次のような利点が得られる。

【０１６９】

①論理装置の動作予測値との比較により、「機能」の確認，「機能」を実現するアーキテクチャの妥当性を設計の初期段階で確認して論理装置の機能検証と性能評価を行なうことができる。

②論理装置の動作予測値との比較により、設計の初期段階に「機能」を正確に実現できたか否かを確認することができる。

【０１７０】

③アーキテクチャに変更が必要になった場合でも、「機能」の記述変更は殆ど行なわずに、リソースやリソースの割り当てルール，スレッドのスケジューリング，リソース要求などを、アーキテクチャの変更に応じて変更するだけで、簡単に設計の再検証を行なうことが可能になり、「機能」の記述の変更による設計コスト増加を回避することができる。

【０１７１】

④「機能」に適したアーキテクチャの探索や、「機能」をアーキテクチャへマッピングする際のアーキテクチャの性能評価，設計初期段階におけるタイミングの検証などが行なえる。

なお、論理装置の動作完了までに必要な一連の「機能」は、複数の逐次的なスレッド群に表されていてもよいし、複数の逐次的または並行に実行されるスレッド群に表されていてもよい。前者の場合は、論理装置の「機能」の依存関係をより明確にした上での動作シミュレーションが可能になり、後者の場合はより複雑な「機能」構成をもった論理装置の動作シミュレーションが可能になる。

【０１７２】

また、上記のリソースマネージャは、リソースの種類に対応して複数設けられていてもよく、例えば、それぞれが管理するリソースを予め規定されたローカルなルールに従ってスレッドに割り当てたり、各リソースマネージャをリソースの依存関係に応じて階層化して、それぞれの管理するリソースと下位階層のリソースマネージャの管理するリソースとの依存関係を考慮してリソースの割り当てを行なったりすることも可能である。

【０１７３】

前者の場合は簡単な記述（構成）でリソースに依存関係のない論理装置のシミュレーションが可能となり、後者の場合はリソースの依存関係をも考慮したグローバルなリソースの割り当て調停が可能となる。

さらに、上記のリソースマネージャは、リソース要求を監視し、その監視結果に基づいて複数スレッド間のリソース要求のデッドロック状態を判定することもでき、この場合は、設計の初期段階においてデッドロック状態の発生可能性を検出することが可能となり、論理装置の機能検証を行なうことが可能となる。

【０１７４】

また、上記のリソースマネージャは、リソース要求によって割り当てられたリソースに対するリード／ライト要求を監視し、その監視結果に基づいて複数スレッドのリソースに対するリード／ライト動作の競合状態を判定することもでき、この場合は、複数のスレッドがリソースに対してシーケンシャルに書き込み動作もしくは読み出し動作を行なう時に、その順序が正しいか否かを判定して、設計

した論理装置が正しく動作するか否かを検証することができる。

【０１７５】

さらに、上記のリソースマネージャは、リソースに対するリソース要求数を監視し、その監視結果に基づいてスレッドのボトルネックを検出することもでき、この場合は、アクセス頻度が最も多いリソースは論理装置のボトルネックになっている可能性があると推測することができ、論理装置の設計初期段階で論理装置の性能検証（ボトルネックの有無の検証）を実現することができる。

【０１７６】

また、上記のリソースマネージャは、リソースに対するリソース要求数を監視し、その監視結果に基づいてリソース要求のブロッキングを検出することもできる。この場合は、リソース要求のブロッキングの発生可能性を検出することで、論理装置の性能を評価することができる。

【０１７７】

さらに、上記のスレッドには、リソースマネージャによって割り当てられたリソースを占有する時間についての予算を与えておいてもよい。このようにすれば、各種「機能」の組み合わせで構成される論理装置が、設計仕様により要求される処理時間に関する性能要件を満たしているか否かを、設計の初期段階にて確認することができる。

【０１７８】

また、上記のスレッドには、上記機能の実行制限時間を与えておいてもよく、このようすれば、設計の初期段階においてリアルタイム性に関する性能要件を論理装置が満足しているかを検証することが可能となる。

さらに、上述した動作シミュレーション方法は、コンピュータを上述のごとく動作させる動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体によって提供されてもよく、このようすれば、コンピュータに上記プログラムを読み取らせれば（インストールすれば）、そのコンピュータを上述したような論理装置の動作シミュレーション方法を実行する装置（論理装置の動作シミュレーション装置）として機能させることができるので、その普及に大きく寄与する。

【図面の簡単な説明】

【図１】

本発明の第１実施形態に係るパーソナルコンピュータなどの計算機（情報処理装置）の構成を示すブロック図である。

【図２】

本発明の第１実施形態に係る動作シミュレーションプログラムを説明するためのオブジェクトモデル図である。

【図３】

図２に示す動作シミュレーションプログラムに基づく計算機の動作（論理装置の動作シミュレーション方法）を説明するための図である。

【図４】

図１に示す計算機の要部に着目した構成を示すブロック図である。

【図５】

第１実施形態に係るスレッドの記述例を示す図である。

【図６】

第１実施形態に係るリソースマネージャの記述例を示す図である。

【図７】

第１実施形態に係る論理装置（ＱｏＳシステム）に必要な「機能」を説明するためのブロック図である。

【図８】

第１実施形態に係る論理装置（ＱｏＳシステム）の設計手順（動作シミュレーション方法）を説明するためのフローチャートである。

【図９】

第１実施形態の第１変形例に係る計算機の要部に着目した構成を示すブロック図である。

【図１０】

第１変形例に係るスレッドの記述例を示す図である。

【図１１】

（Ａ）及び（Ｂ）はそれぞれ第１変形例に係るスレッドのデッドロックの検出

を説明するためのリソース割付けグラフである。

【図１２】

第１実施形態の第２変形例に係る計算機の要部に着目した構成を示すブロック図である。

【図１３】

第２変形例に係るリソース要求の記述例を示す図である。

【図１４】

第２変形例に係るリソースの記述例を示す図である。

【図１５】

第１実施形態の第３変形例に係る計算機の要部に着目した構成を示すブロック図である。

【図１６】

第３変形例に係るリソースの記述例を示す図である。

【図１７】

第１実施形態の第４変形例に係る計算機の要部に着目した構成を示すブロック図である。

【図１８】

第４変形例に係るリソースの記述例を示す図である。

【図１９】

第１実施形態の第５変形例に係る論理装置の設計手順（動作シミュレーション方法）を説明するためのフローチャートである。

【図２０】

第５変形例に係るスレッドの記述例を示す図である。

【図２１】

第１実施形態の第６変形例に係る論理装置の設計手順（動作シミュレーション方法）を説明するためのフローチャートである。

【図２２】

第６変形例に係るスレッドの記述例を示す図である。

【図２３】

第１実施形態の第７変形例に係るスレッドの生成方法を説明するための模式図である。

【図２４】

第７変形例に係るスレッドの記述例を示す図である。

【図２５】

第１実施形態の第８変形例に係るスレッドの生成方法を説明するための模式図である。

【図２６】

第８変形例に係るスレッドの記述例を示す図である。

【図２７】

第１実施形態の第９変形例に係る計算機の要部に着目した構成を示すブロック図である。

【図２８】

第９変形例に係るリソースの記述例を示す図である。

【図２９】

第１実施形態の第１０変形例に係る計算機の要部に着目した構成を示すブロック図である。

【図３０】

（Ａ）及び（Ｂ）はいずれも従来の論理装置の設計手順を説明するためのフローチャートである。

【図３１】

従来の論理装置の設計手順による課題を説明するための図である。

【符号の説明】

１　計算機（情報処理装置；論理装置の動作シミュレーション装置）

２　計算機本体

３　ディスプレイ（表示装置；外部装置）

４　ＣＰＵ

５　主記憶部（メモリ）

６　二次記憶装置（ハードディスク）

７　フロッピーディスク（ＦＤ）ドライブ

８　内部バス

９　フロッピーディスク（コンピュータ読み取り可能な記録媒体）

１０，５１　論理装置の動作シミュレーションプログラム

１１　スレッドマネージャ

１２，１２－１～１２－ｎ　リソースマネージャ

１３　スレッド

１４，１４－１～１４－ｎ　リソース（ハードウェア資源）

１５　テストベンチ

１６　テストデータ

１７　入力キュー

１８　実行待ちキュー

１９　要求

２０　リソース要求キュー

２１　回答

２２　リソース回答キュー

３１ａ，３１ａ′　要求メソッド

３１ｂ，３１ｂ′　解放メソッド

３１ｃ，３１ｃ′　調停メソッド

４１　パケット

４２　クラス分け機能

４３　キューイング機能

４４　デキュー機能

４５　キュー

１１０　スレッド生成手段

１１１　リソース要求手段

１１２　スレッド制御手段

１２１　リソース割り当て手段

１２２　デッドロック検出機構

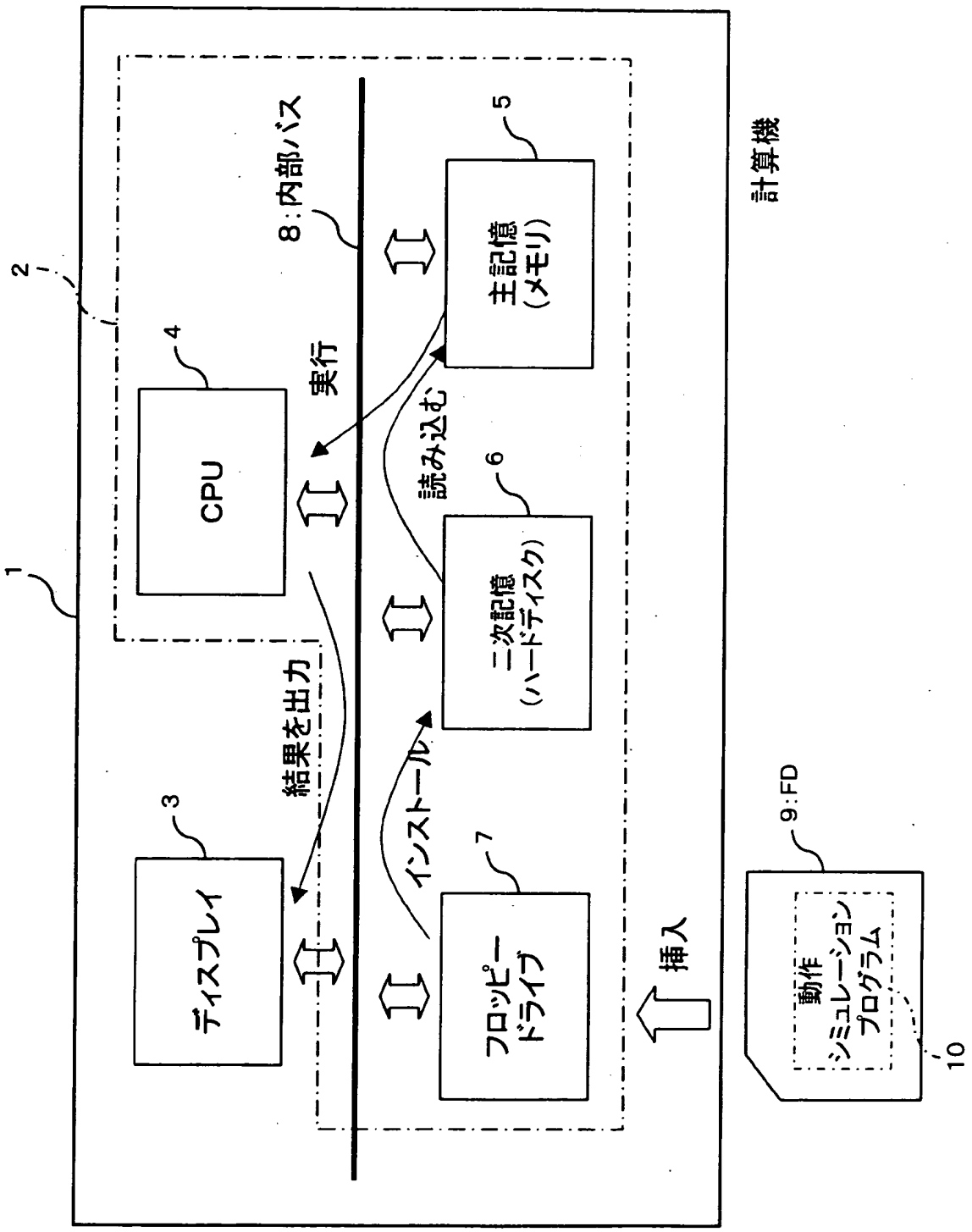
１２３　リード／ライト検出機構

１２４　ボトルネック検出機構

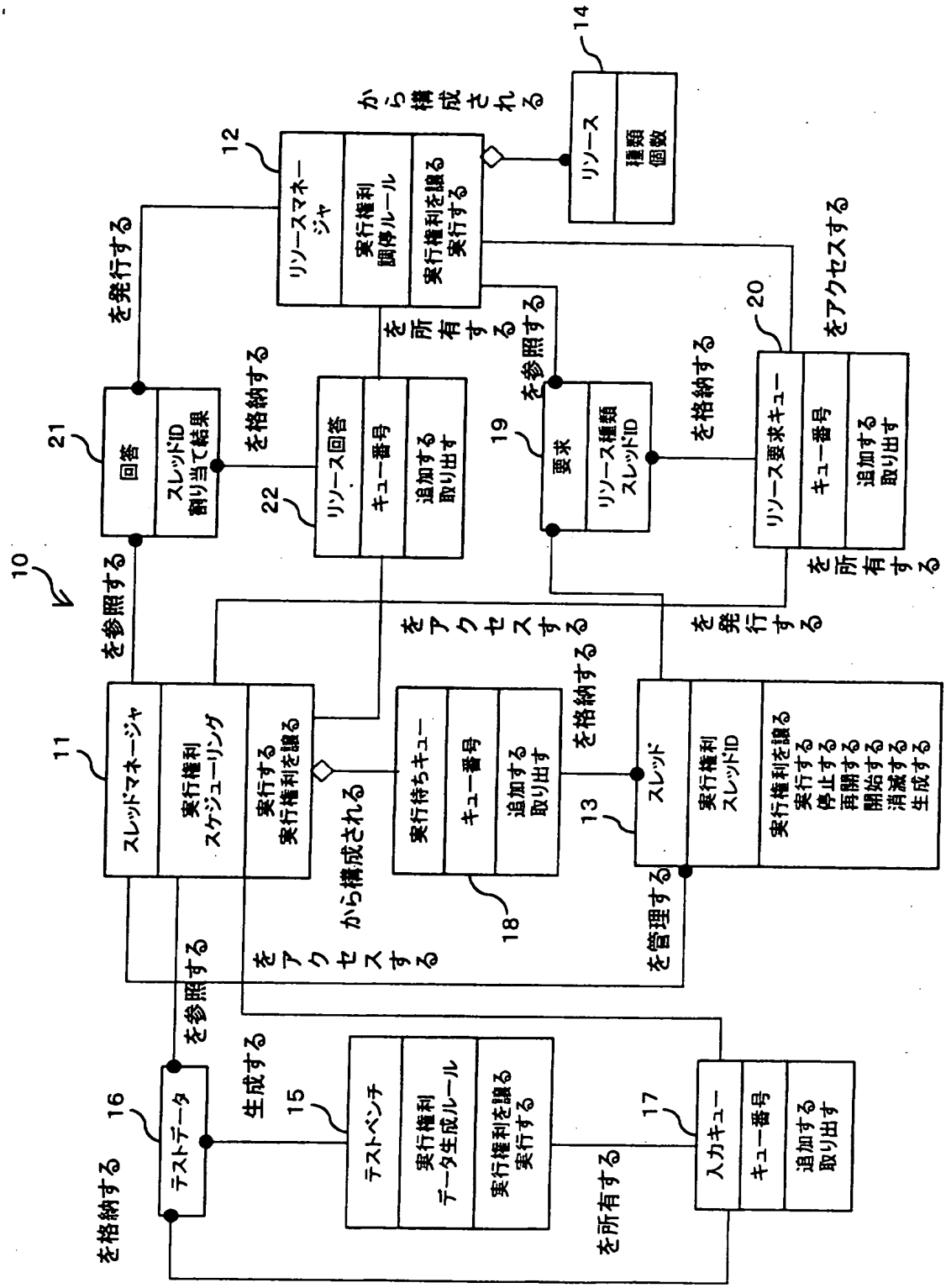
１２５　ブロッキング検出機構

【書類名】　　図面

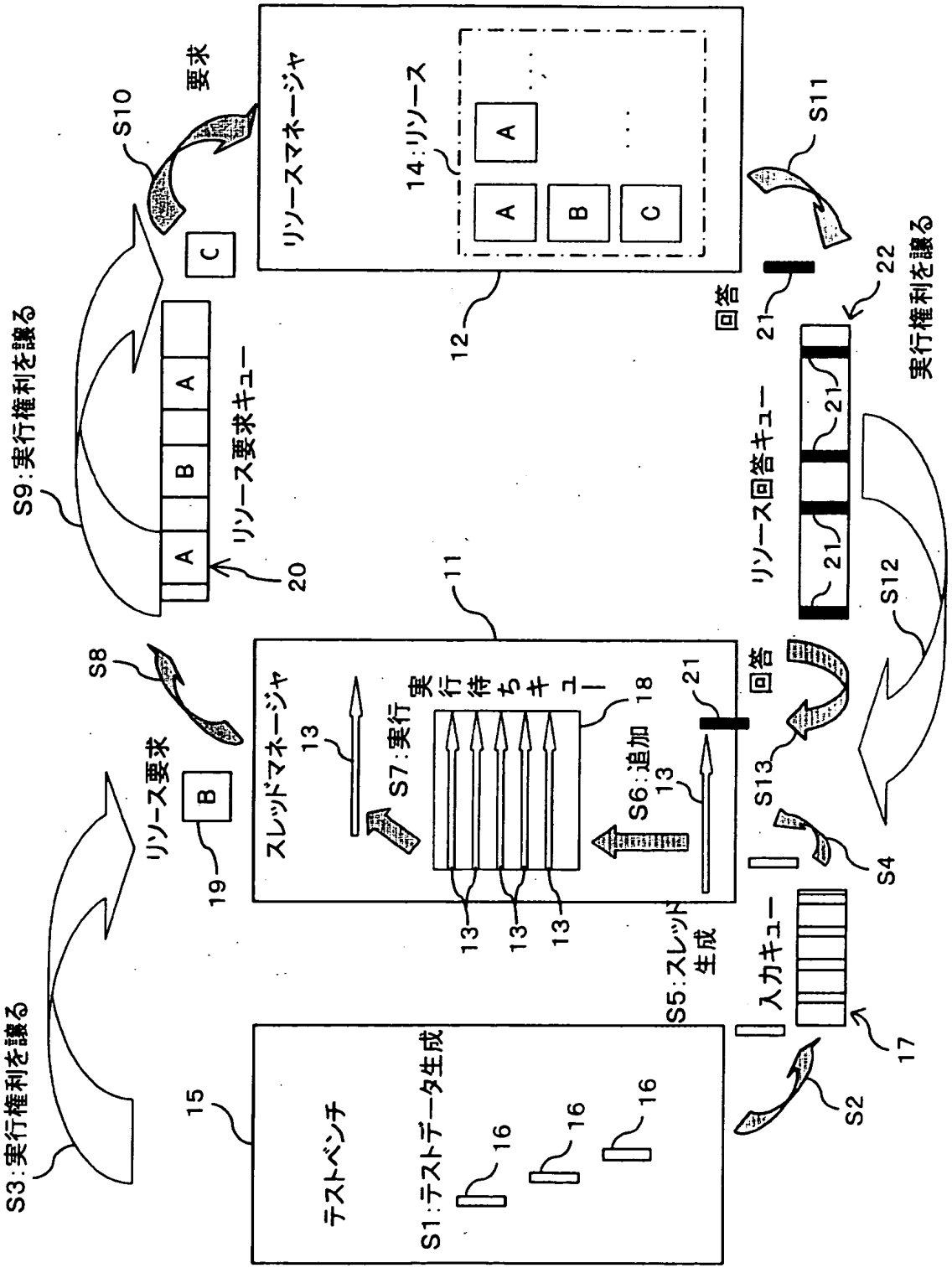
【図１】


計算機
1
2
8:内部バス
5
主記憶
(メモリ)
4
CPU
実行
読み込む
6
二次記憶
(ハードディスク)
結果を出力
3
ディスプレイ
インストール
7
フロッピー
ドライブ
挿入
9:FD
動作
シミュレーション
プログラム
10

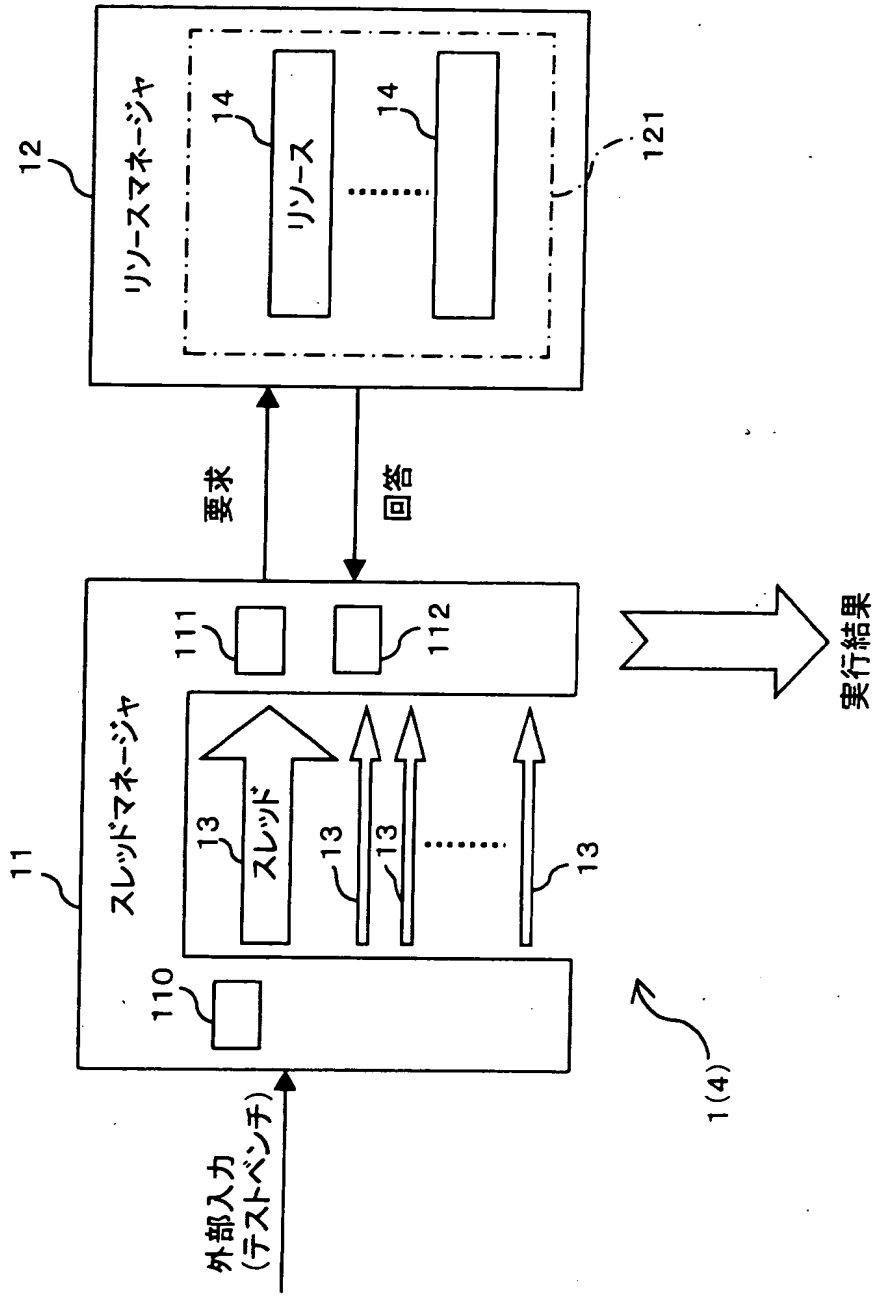
【図２】


10
11
スレッドマネージャ
実行権利
スケジューリング
実行する
実行権利を譲る
12
リソースマネージャ
実行権利
調停ルール
実行権利を譲る
実行する
14
リソース
種類
個数
から構成される
を発行する
を所有する
を参照する
を格納する
21
回答
スレッドID
割り当て結果
22
リソース回答
キュー番号
追加する
取り出す
19
要求
リソース種類
スレッドID
20
リソース要求キュー
キュー番号
追加する
取り出す
をアクセスする
を所有する
を発行する
を参照する
を格納する
をアクセスする
から構成される
実行待ちキュー
キュー番号
追加する
取り出す
18
13
スレッド
実行権利
スレッドID
実行権利を譲る
実行する
停止する
再開する
開始する
消滅する
生成する
を管理する
をアクセスする
を参照する
16
テストデータ
生成する
15
テストベンチ
実行権利
データ生成ルール
実行権利を譲る
実行する
17
入力キュー
キュー番号
追加する
取り出す
を格納する
を所有する

【図３】


テストベンチ
15
S1:テストデータ生成
16
S2
入力キュー
17
S3:実行権利を譲る
リソース要求
B
19
S4
スレッドマネージャ
13
S5:スレッド生成
S6:追加
S7:実行
実行待ちキュー
18
21
S8
S9:実行権利を譲る
リソース要求キュー
A
B
C
20
S10
要求
リソースマネージャ
14:リソース
12
S11
回答
21
リソース回答キュー
22
実行権利を譲る
S12
S13
回答
11

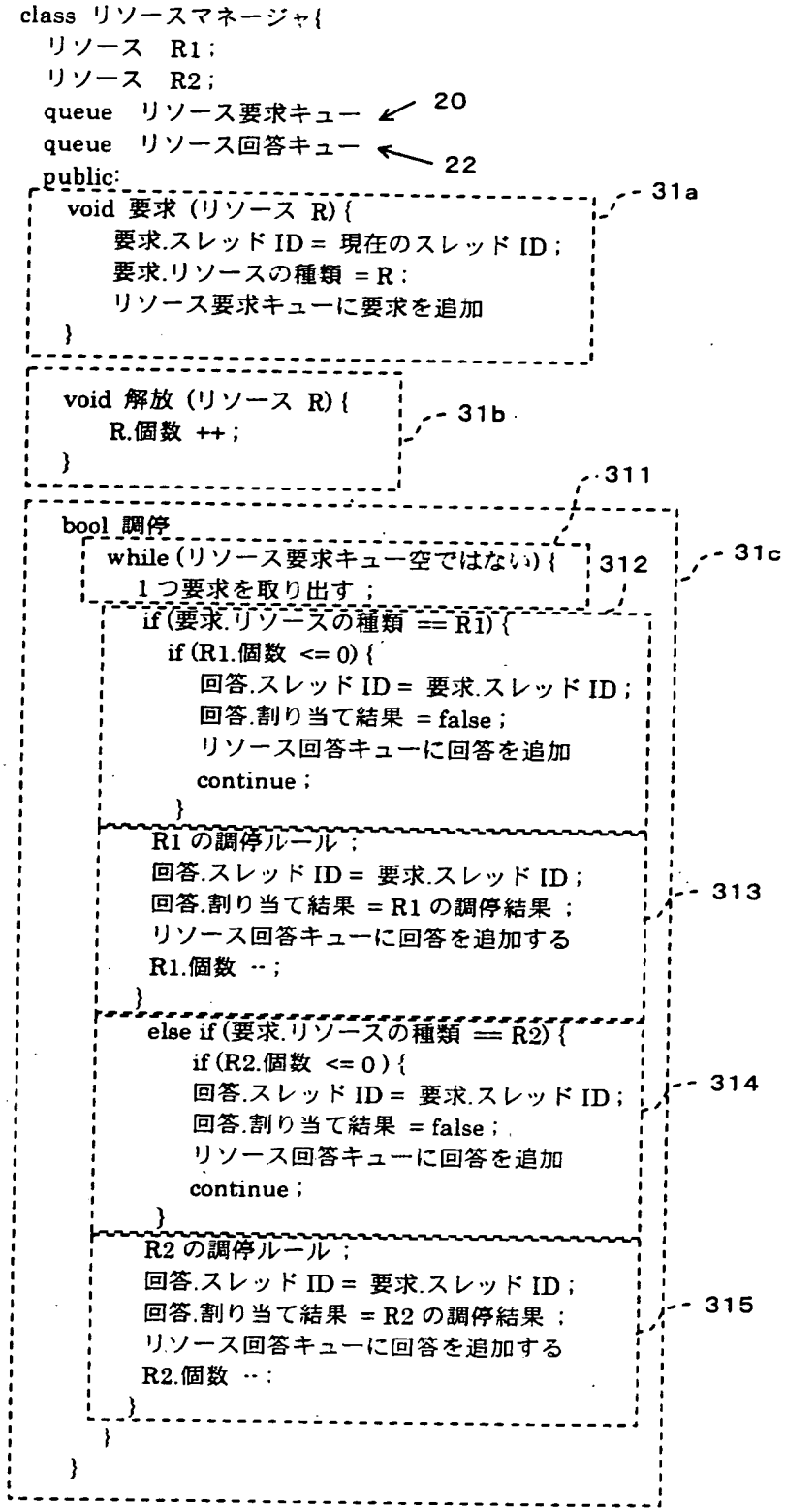
【図４】


12
リソースマネージャ
14
リソース
14
121
要求
回答
111
112
実行結果
11
スレッドマネージャ
13
スレッド
13
13
13
110
1(4)
外部入力
(テストベンチ)

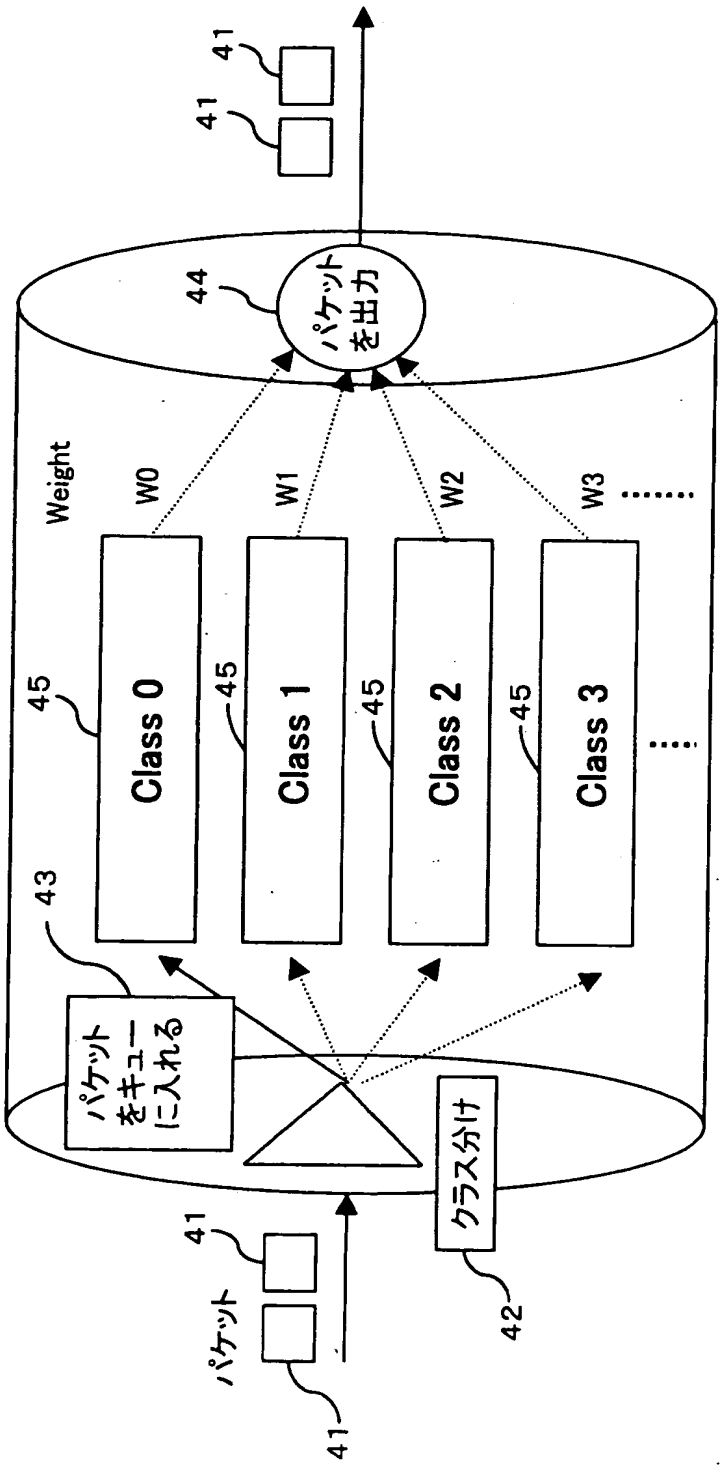
【図５】

```
class スレッド{
  …
  void 実行(){
    処理A();
    処理B();
    処理C();
  }
}
```

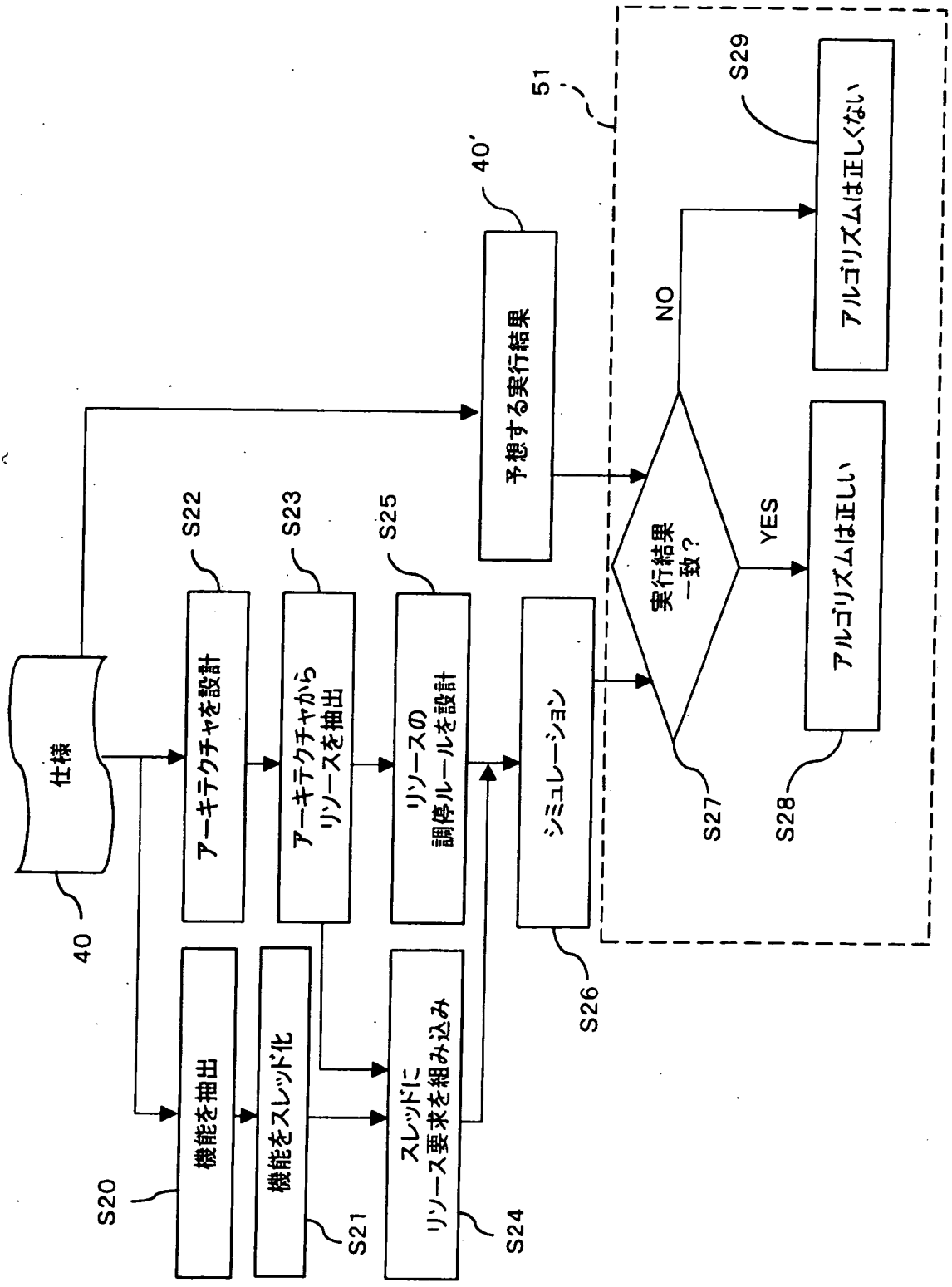
【図６】


class リソースマネージャ{
リソース R1;
リソース R2;
queue リソース要求キュー
queue リソース回答キュー
20
22
public:
void 要求 (リソース R){
要求.スレッドID = 現在のスレッドID;
要求.リソースの種類 = R;
リソース要求キューに要求を追加
}
31a
void 解放 (リソース R){
R.個数 ++;
}
31b
311
bool 調停
while (リソース要求キュー空ではない){
1つ要求を取り出す;
312
31c
if (要求.リソースの種類 == R1){
if (R1.個数 <= 0){
回答.スレッドID = 要求.スレッドID;
回答.割り当て結果 = false;
リソース回答キューに回答を追加
continue;
}
R1の調停ルール;
回答.スレッドID = 要求.スレッドID;
回答.割り当て結果 = R1の調停結果;
リソース回答キューに回答を追加する
R1.個数 --;
313
}
else if (要求.リソースの種類 == R2){
if (R2.個数 <= 0){
回答.スレッドID = 要求.スレッドID;
回答.割り当て結果 = false;
リソース回答キューに回答を追加
continue;
}
314
R2の調停ルール;
回答.スレッドID = 要求.スレッドID;
回答.割り当て結果 = R2の調停結果;
リソース回答キューに回答を追加する
R2.個数 --;
315
}
}
}

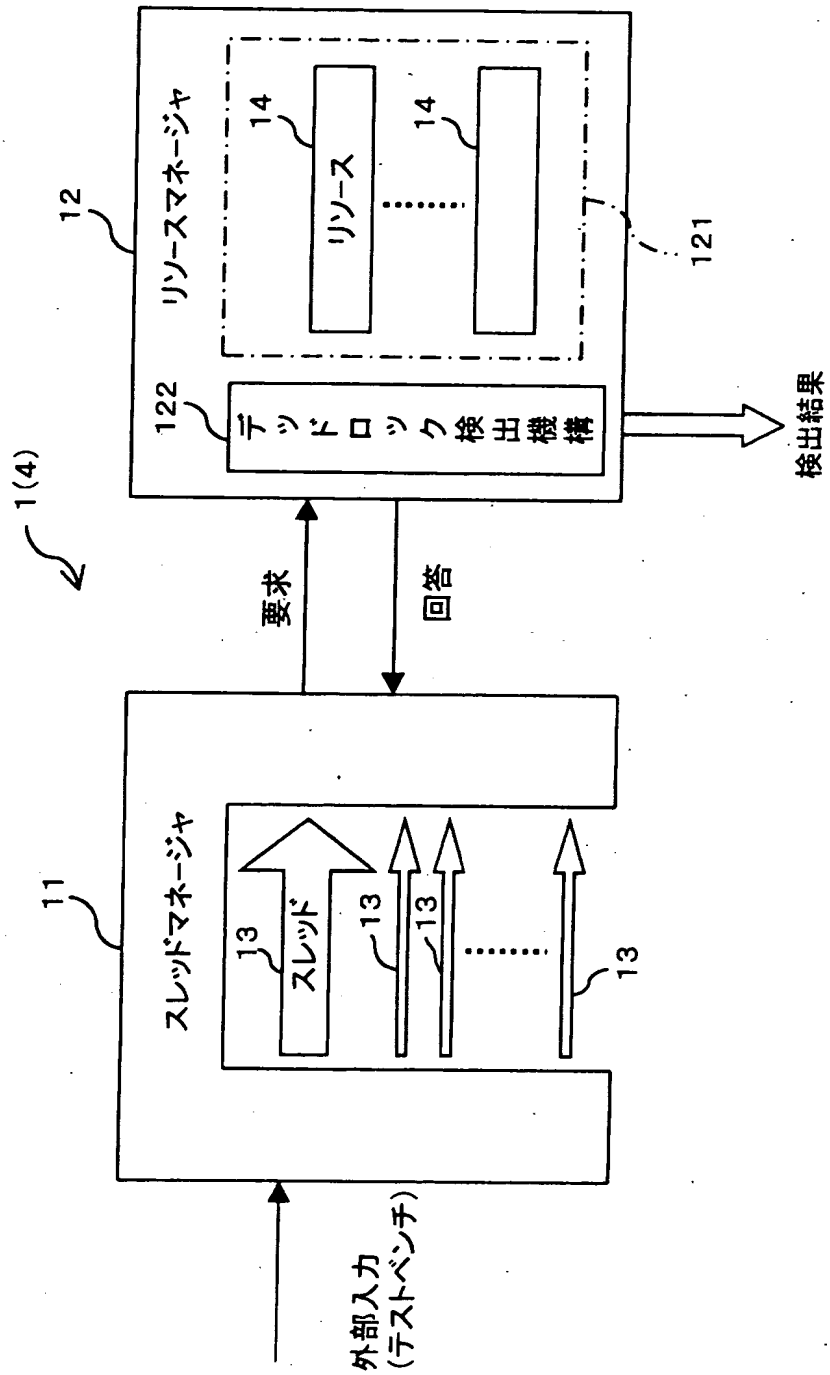
【図７】


41
41
44
パケットを出力
Weight
W0
W1
W2
W3
45
Class 0
45
Class 1
45
Class 2
45
Class 3
43
パケットをキューに入れる
クラス分け
42
41
パケット
41

【図８】


仕様
40
アーキテクチャを設計
S22
アーキテクチャから
リソースを抽出
S23
リソースの
調停ルールを設計
S25
シミュレーション
S26
予想する実行結果
40'
51
実行結果
一致？
S27
NO
YES
アルゴリズムは正しくない
S29
アルゴリズムは正しい
S28
機能を抽出
S20
機能をスレッド化
S21
スレッドに
リソース要求を組み込み
S24

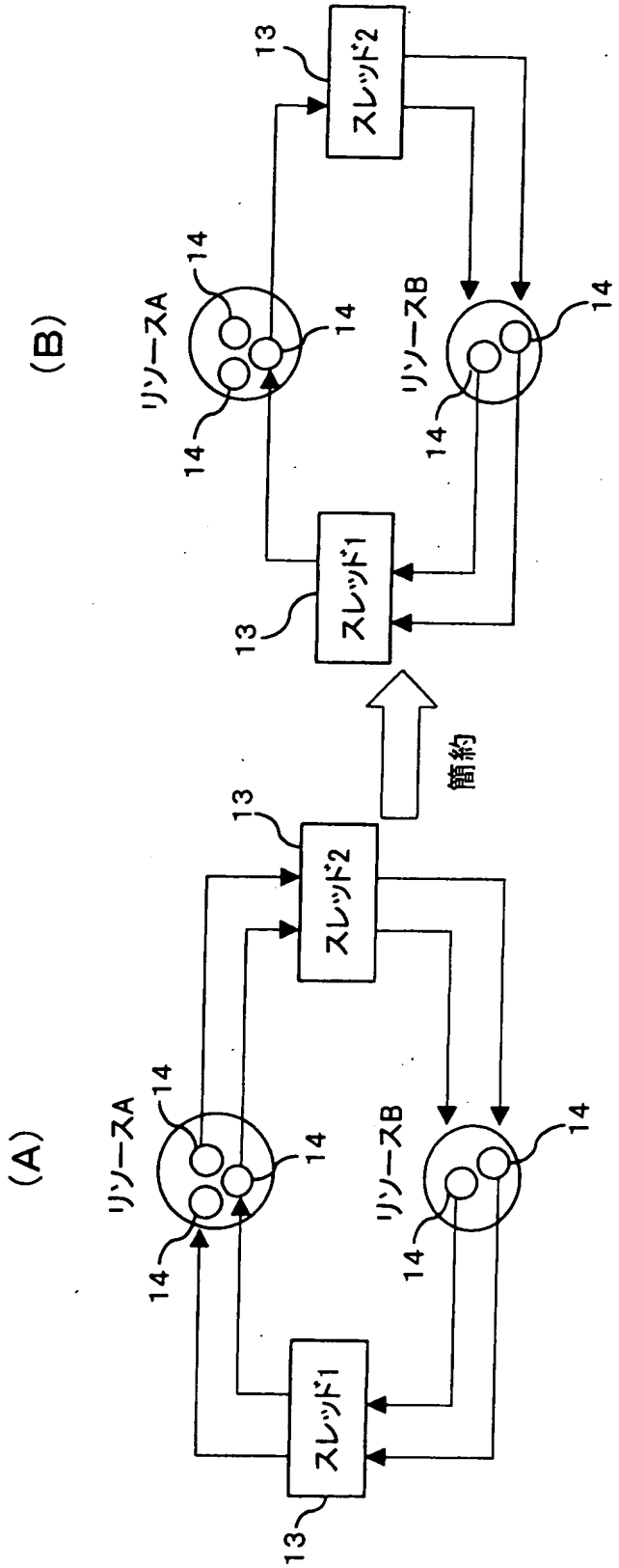
【図９】


外部入力
(テストベンチ)
スレッドマネージャ
11
スレッド
13
13
13
13
要求
回答
1(4)
リソースマネージャ
12
リソース
14
14
121
デッドロック検出機構
122
検出結果

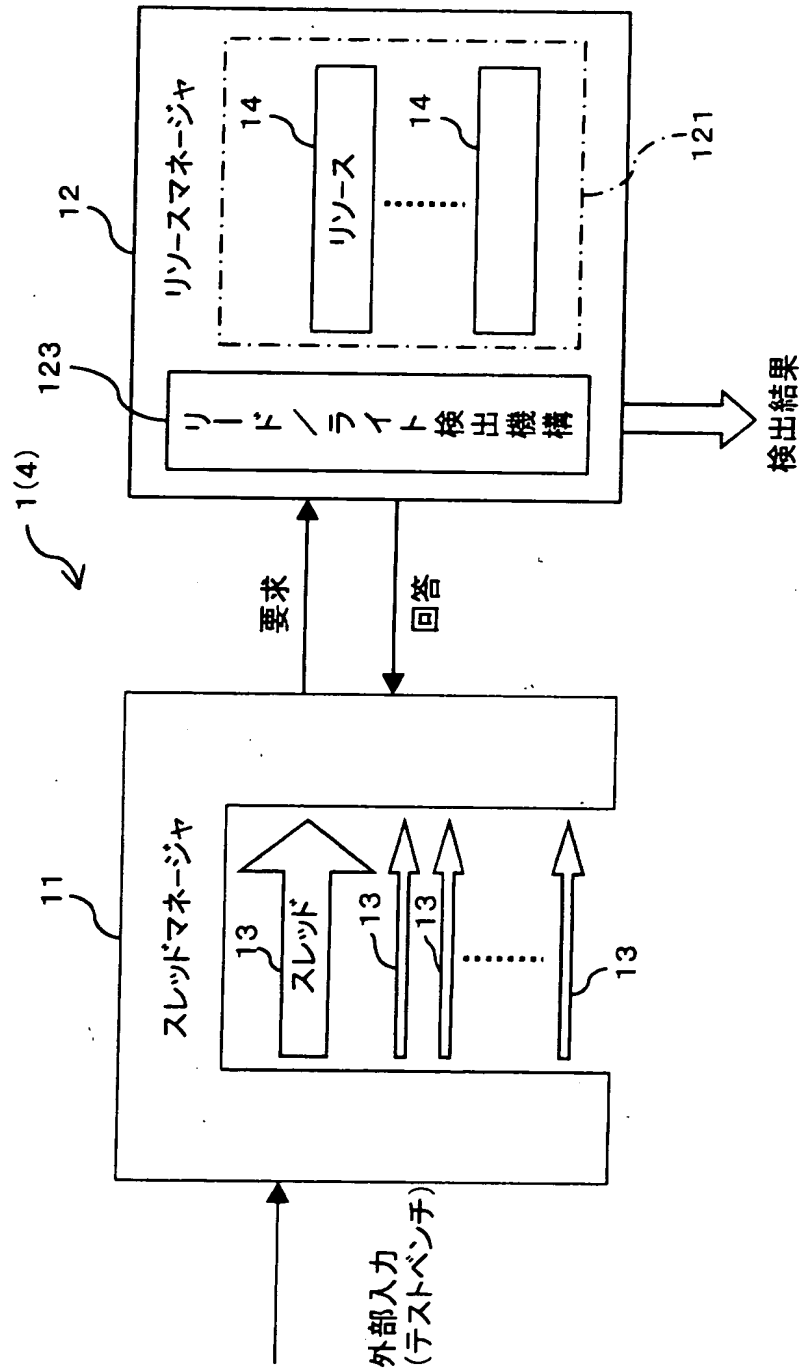
【図１０】

```
class スレッド{
  …
  void 実行(){
    リソースA.要求(2);
    処理;
    …
リソースB.要求(2);
リソースA.解放(2);
    処理;
    …
    リソースA.要求(2);
    リソースB.解放(2);
    処理;
    …
    リソースA.解放(2);
  …
  }
}
```

【図１１】



【図１２】


リソースマネージャ
12
14
リソース
14
121
123
リード／ライト検出機構
検出結果
1(4)
要求
回答
スレッドマネージャ
11
13
スレッド
13
13
13
外部入力
(テストベンチ)

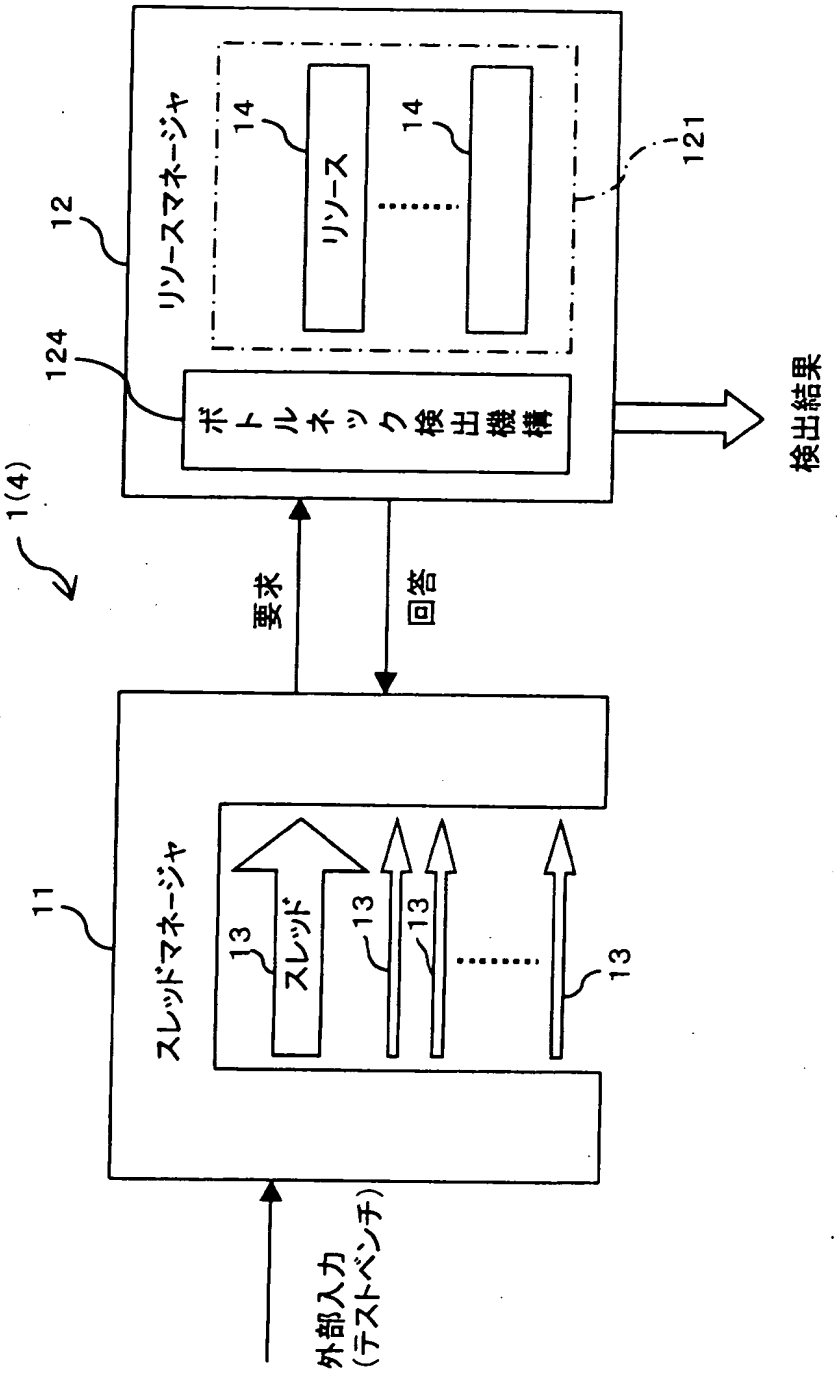
【図１３】

```
class 要求{
 unsigned int スレッドID;
 int 要求個数;
 int リード／ライトフラグ;
}
```

【図１４】

```
class リソース A: public リソース{
int CurrentFlag = 0;
 ...
 void 要求(int n, bool ReadWriteFlag( ){
        要求.スレッドID = 現在のスレッドID;
        要求.リソース数 = n;
        要求.リード／ライトフラグ = ReadWriteFlag;
リソース要求キューに要求を追加する。
}
void 解放(int n){                          316
     ...
     CurrentFlag = 0;
}
 ...
bool 調停( ){
     ...                                          317
 while (リソース要求キュー空ではない){
       １つ要求を取り出す;
 if (CurrentFlag !=0 && 要求.リード／ライトフラグ
   != CurrentFlag){
         error("リード／ライトエラーの発生可能性がある！");
       }
       CurrentFlag == 要求.リード／ライトフラグ;
     }
    ...
  }
  ...
}
```

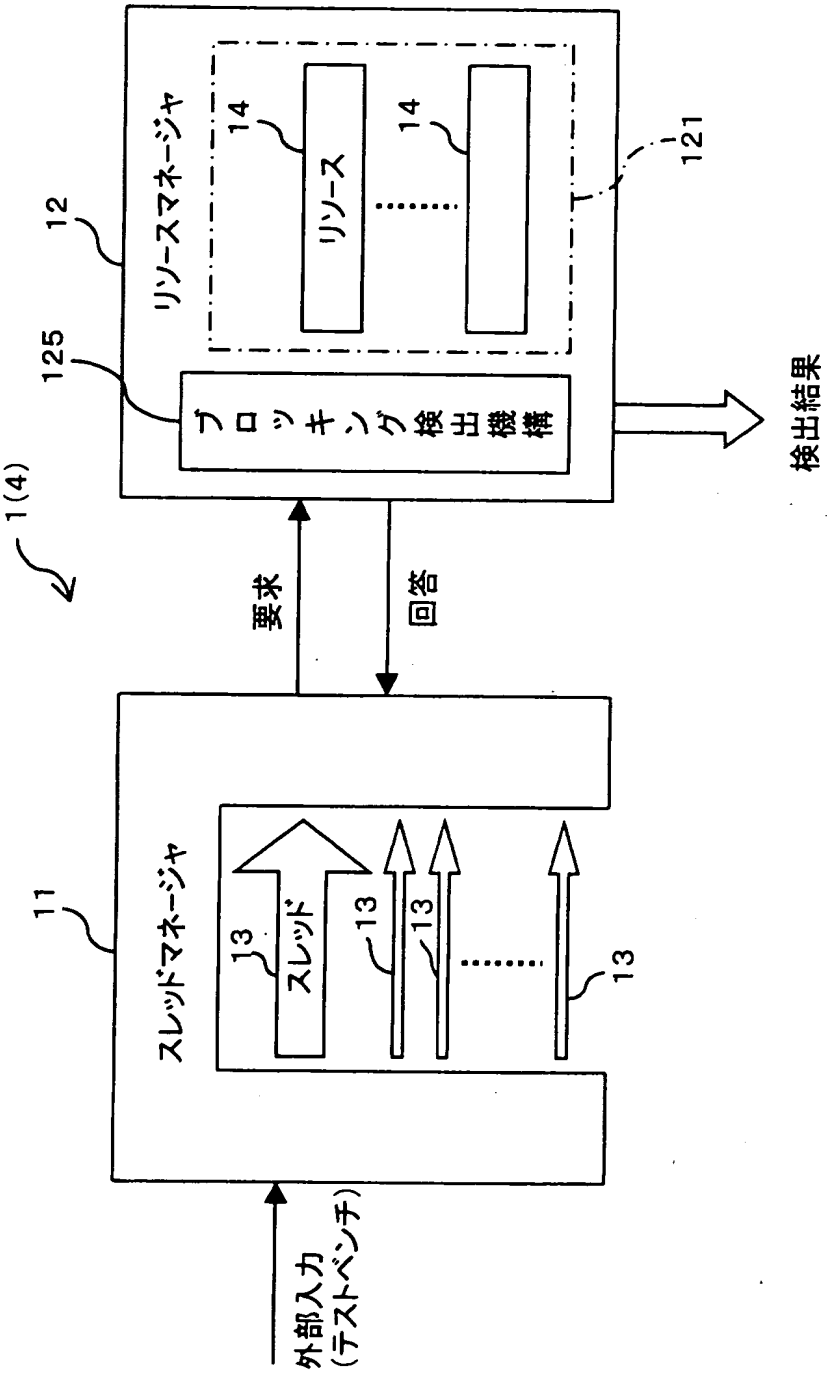
【図１５】


リソースマネージャ
12
14
リソース
14
121
124
ボトルネック検出機構
検出結果
1(4)
要求
回答
スレッドマネージャ
11
13
スレッド
13
13
13
外部入力
(テストベンチ)

【図１６】

```
class リソース {
  int アクセス数 =0;
  …
  void 要求(int n){
     アクセス数++;
  }
  …
  int 要求合計(){
     return アクセス数;
  }
}
```

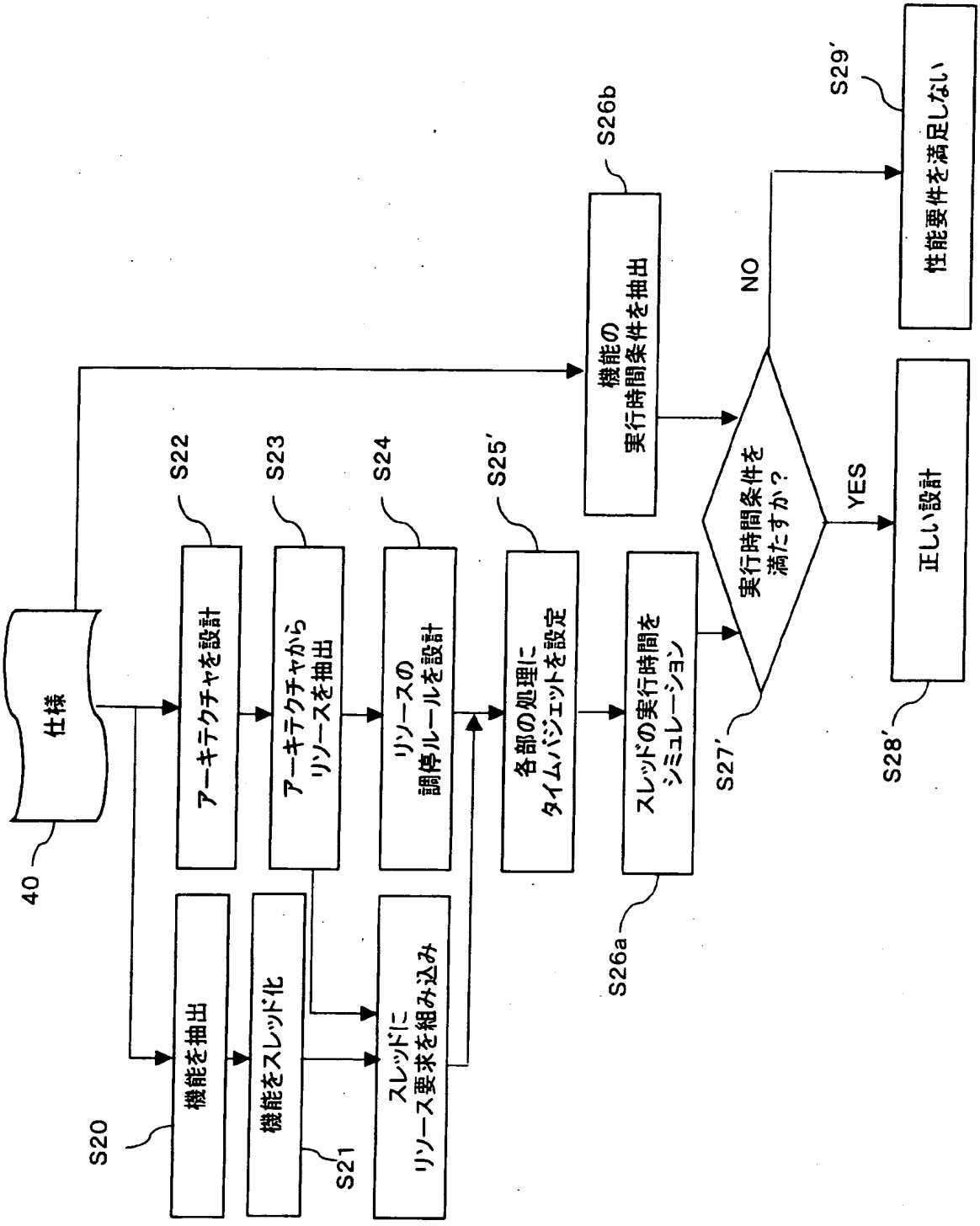
【図１７】


リソースマネージャ
12
14
リソース
14
121
125
ブロッキング検出機構
検出結果
1(4)
要求
回答
スレッドマネージャ
11
13
スレッド
13
13
13
外部入力
(テストベンチ)

【図１８】

```
class リソース A{
      …
      bool 調停( ){
     while (リソース要求キュー空ではない){
        一つ要求を取り出す;
if (個数＜０){
  error("リソースＡの要求はブロッキングする必要がある");
}
      }
      …
 }
```

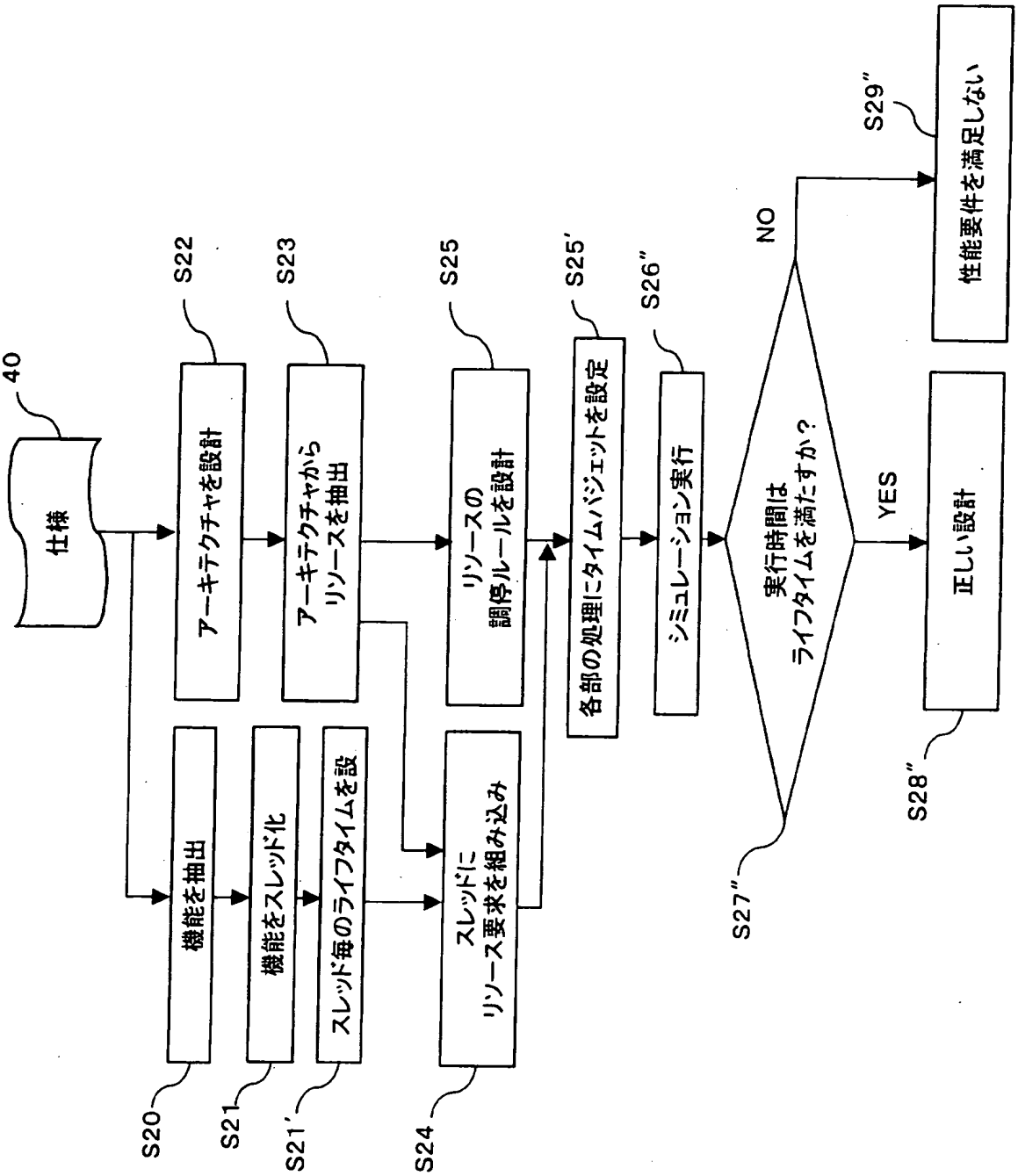
【図１９】


仕様
40
アーキテクチャを設計
S22
アーキテクチャから
リソースを抽出
S23
リソースの
調停ルールを設計
S24
各部の処理に
タイムバジェットを設定
S25′
スレッドの実行時間を
シミュレーション
S26a
機能の
実行時間条件を抽出
S26b
実行時間条件を
満たすか？
S27′
NO
YES
正しい設計
S28′
性能要件を満足しない
S29′
機能を抽出
S20
機能をスレッド化
S21
スレッドに
リソース要求を組み込み

【図２０】

```
class スレッド{
  …
  void 実行(){
    リソースA.要求(1);
    処理１;
    delay(処理１のタイムバジェット);
    リソースA.解放(１);
    リソースB.要求(１);
    処理２:
    delay(処理２のタイムバジェット);
    リソースB.解放(１);
    …
  }
}
```

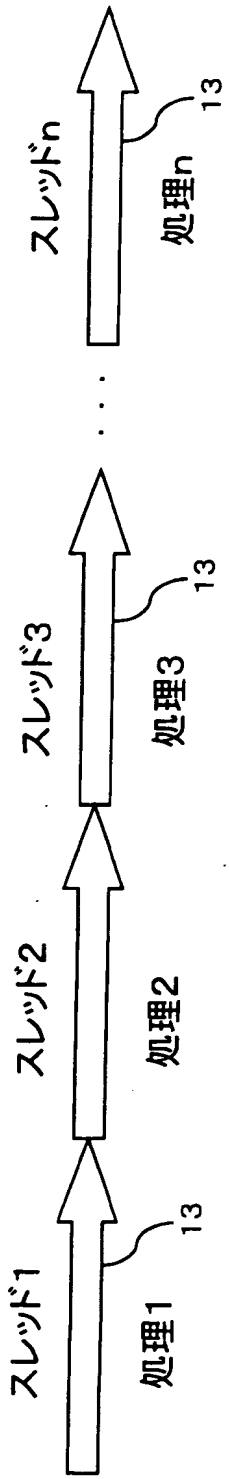
【図２１】


40
仕様
S22
アーキテクチャを設計
S23
アーキテクチャから
リソースを抽出
S25
リソースの
調停ルールを設計
S25'
各部の処理にタイムバジェットを設定
S26"
シミュレーション実行
実行時間は
ライフタイムを満たすか？
NO
S29"
性能要件を満足しない
YES
正しい設計
S28"
S27"
S20
機能を抽出
S21
機能をスレッド化
S21'
スレッド毎のライフタイムを設
S24
スレッドに
リソース要求を組み込み

【図２２】

```
class スレッド {
    long ライフタイム;
    long 遅延;
    void 実行(){
        …
    }
    bool JugdeLifeTime(){
        if(ライフタイム < 遅延)
        return false;
        return true;
    }
}
```

【図２３】



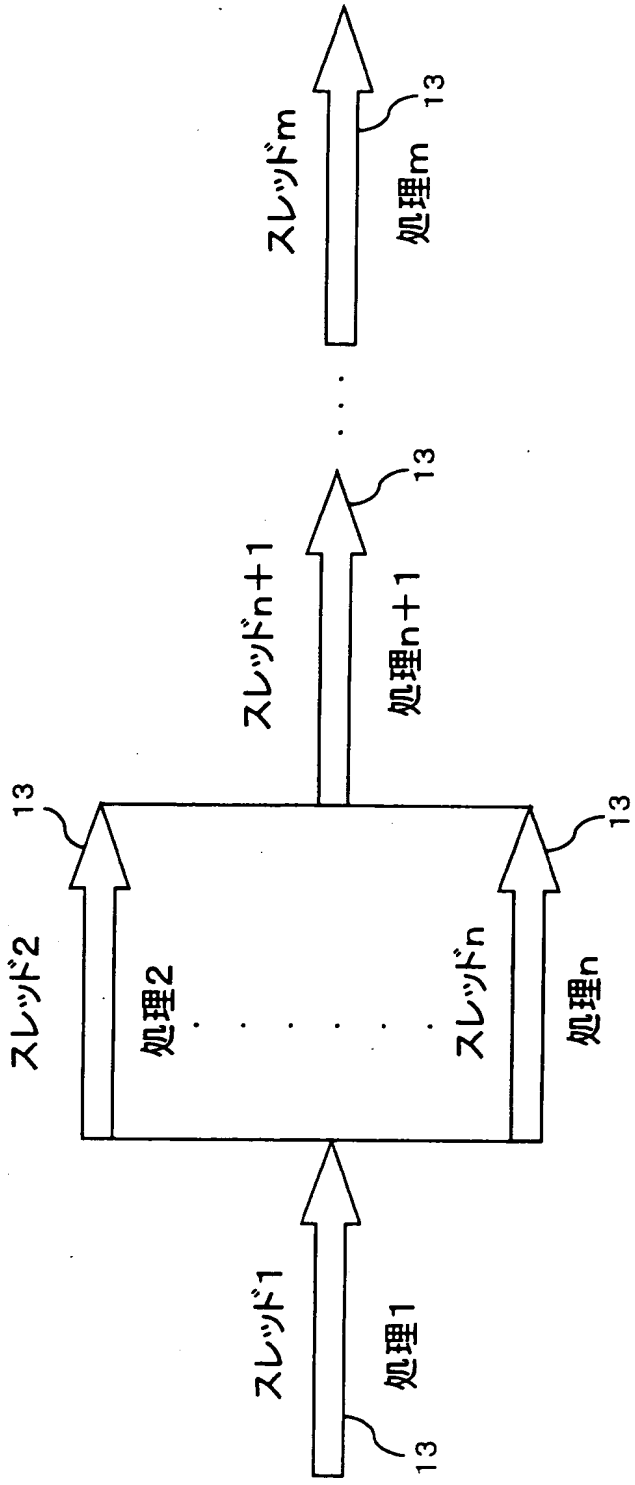
【図２４】

```
class スレッド{
  …
  void 実行(){
    スレッド 1.生成();
    スレッド 1.終了を待つ();
    スレッド 2.生成();
    スレッド 2.終了を待つ();
    スレッド 3.生成();
    スレッド 3.終了を待つ();
  }
}

class スレッド 1: public スレッド{
  …
    void 実行(){
    処理 1();
   }
  …
}
class スレッド 2: public スレッド{
  …
  void 実行(){
        処理 2();
    }
   …
  }

スレッド 3: public スレッド{
  …
  void 実行(){
     処理 3();
  }
  …
}
```

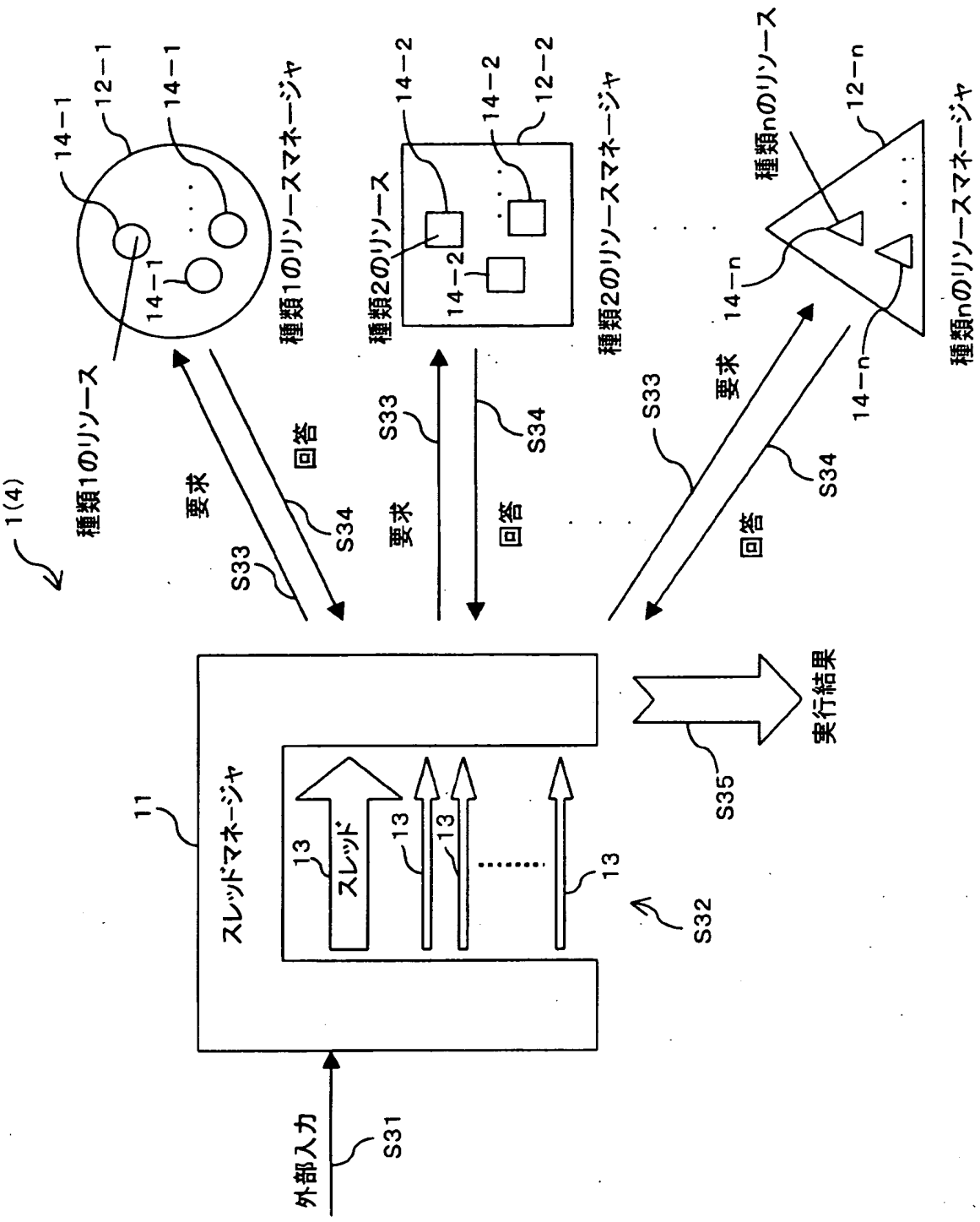
【図２５】


スレッドm
処理m
13
スレッドn+1
処理n+1
13
スレッド2
処理2
スレッドn
処理n
13
13
スレッド1
処理1
13

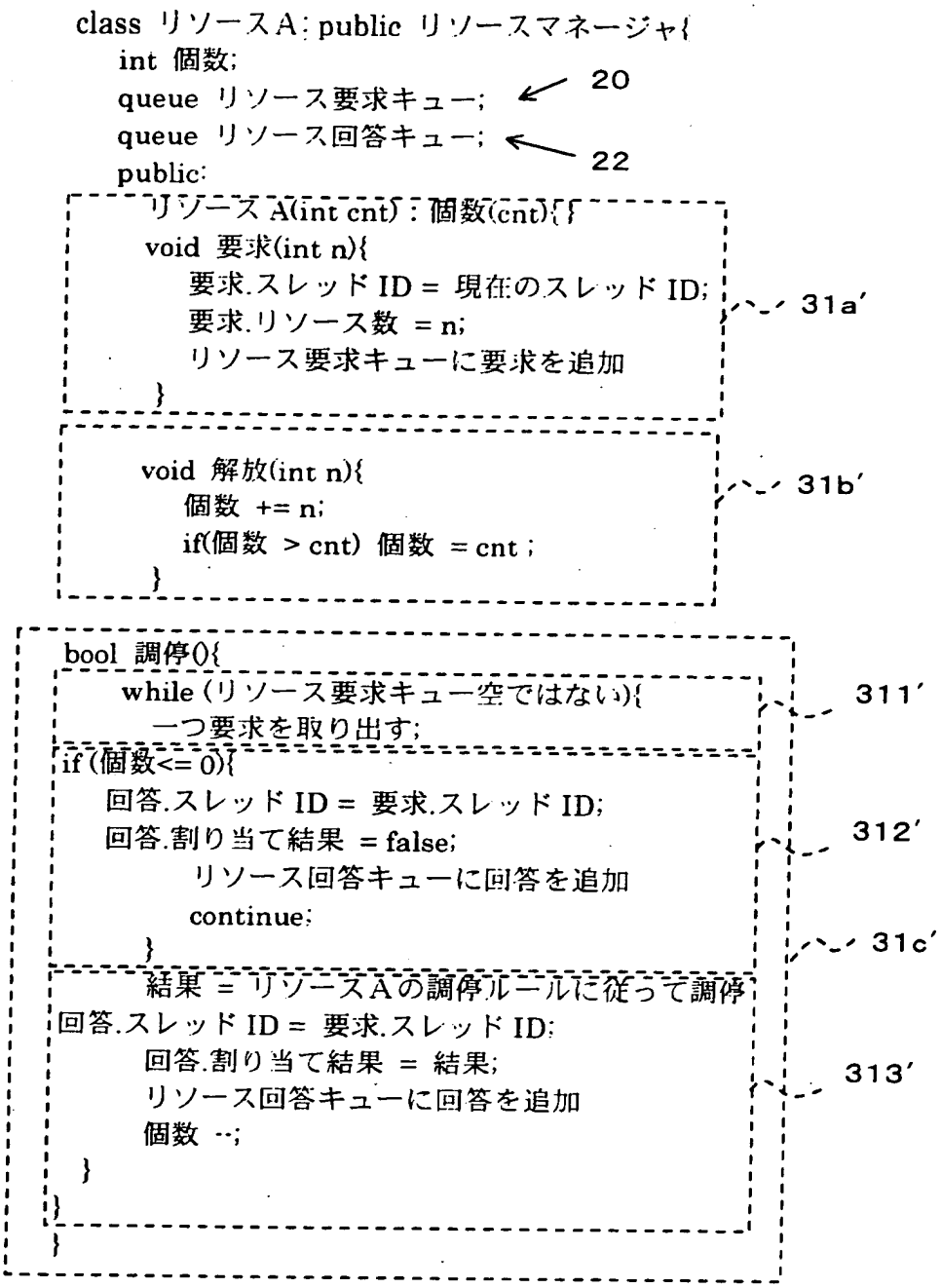
【図２６】

```
class スレッド{
  ...
  void 実行(){
    スレッドA.生成();
    スレッドB.生成();
    スレッドA.終了を待つ();
    スレッドB.終了を待つ();
    スレッドC.生成();
    スレッドC.終了を待つ();
  }
}
```

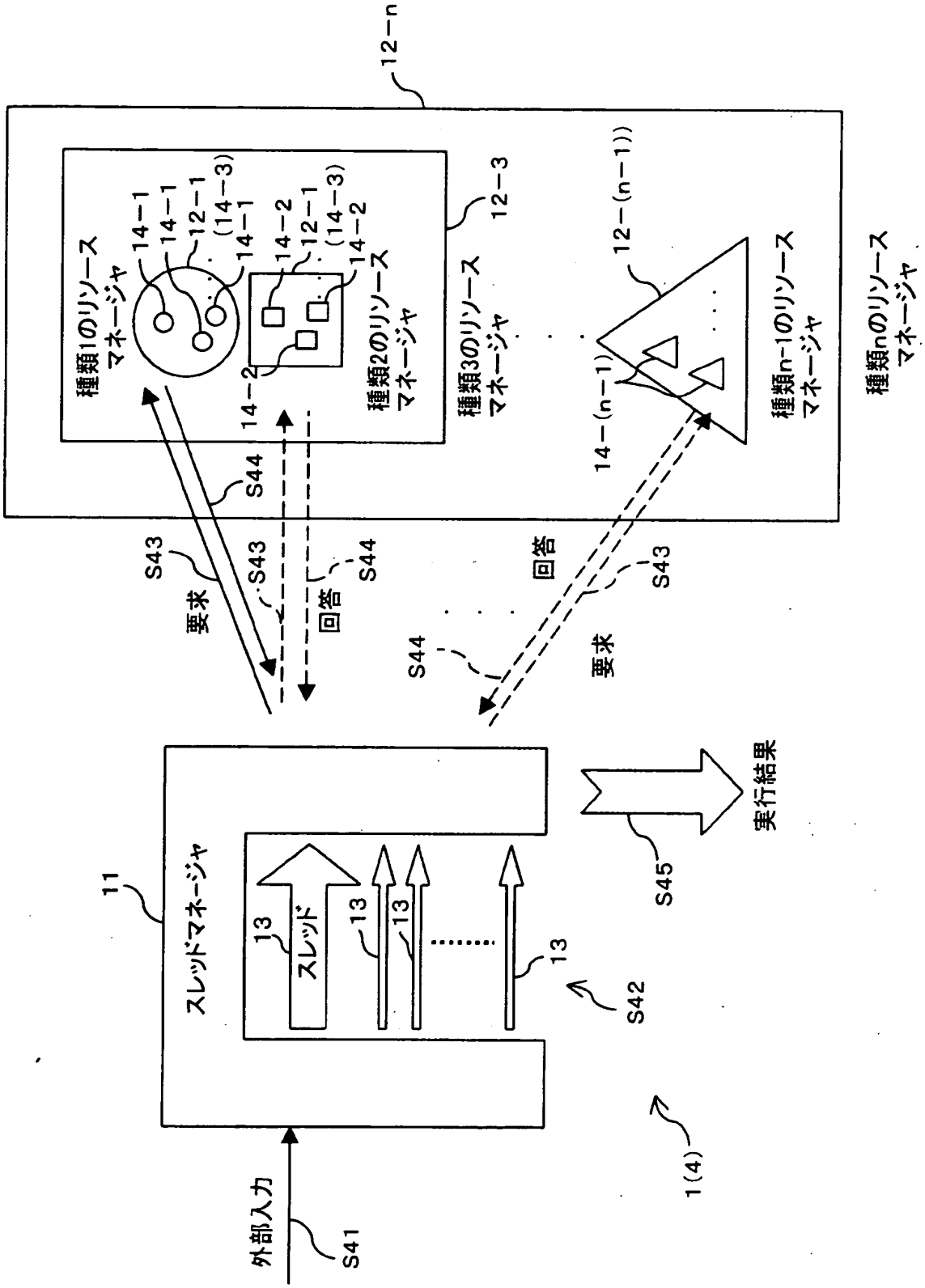
【図２７】


1(4)
11
スレッドマネージャ
13
スレッド
外部入力
S31
S32
S33
S34
S35
実行結果
種類1のリソース
種類1のリソースマネージャ
14－1
12－1
要求
回答
種類2のリソース
種類2のリソースマネージャ
14－2
12－2
種類nのリソース
種類nのリソースマネージャ
14－n
12－n

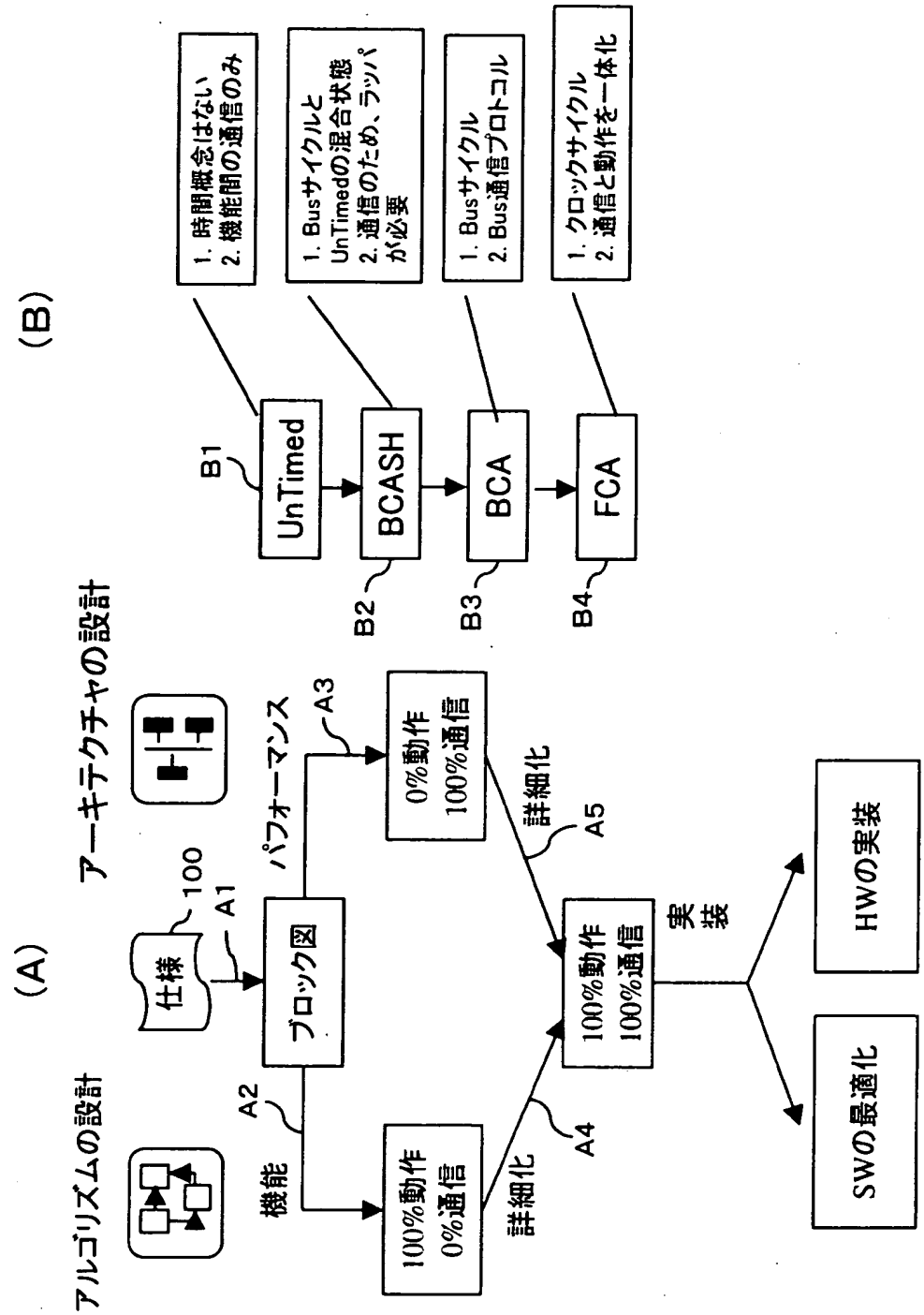
【図２８】


class リソースA: public リソースマネージャ{
int 個数;
queue リソース要求キュー;
20
queue リソース回答キュー;
22
public:
リソースA(int cnt) : 個数(cnt){}
void 要求(int n){
要求.スレッドID = 現在のスレッドID;
要求.リソース数 = n;
リソース要求キューに要求を追加
}
31a'
void 解放(int n){
個数 += n;
if(個数 > cnt) 個数 = cnt ;
}
31b'
bool 調停(){
while (リソース要求キュー空ではない){
311'
一つ要求を取り出す;
if (個数<= 0){
回答.スレッドID = 要求.スレッドID;
回答.割り当て結果 = false;
312'
リソース回答キューに回答を追加
continue;
}
31c'
結果 = リソースAの調停ルールに従って調停
回答.スレッドID = 要求.スレッドID;
回答.割り当て結果 = 結果;
313'
リソース回答キューに回答を追加
個数 --;
}
}
}

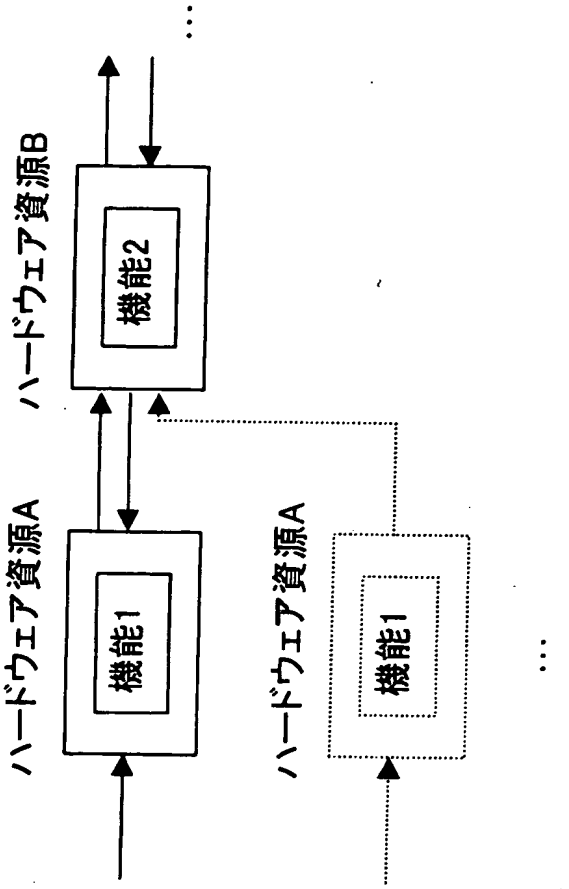
【図２９】


12−n
種類1のリソース
マネージャ
14−1
14−1
12−1
(14−3)
14−1
14−2
12−1
(14−3)
14−2
種類2のリソース
マネージャ
12−3
種類3のリソース
マネージャ
12−(n−1))
14−(n−1)
種類n−1のリソース
マネージャ
種類nのリソース
マネージャ
14−2
S44
S43
要求
S43
回答
S44
S44
回答
S43
要求
11
スレッドマネージャ
13
スレッド
13
13
13
S42
S45
実行結果
外部入力
S41
1(4)

【図３０】


(A)
アルゴリズムの設計
アーキテクチャの設計
仕様
100
A1
ブロック図
A2
機能
パフォーマンス
A3
100%動作
0%通信
詳細化
A4
0%動作
100%通信
詳細化
A5
100%動作
100%通信
実装
HWの実装
SWの最適化
(B)
1. 時間概念はない
2. 機能間の通信のみ
1. Busサイクルと
UnTimedの混合状態
2. 通信のため、ラッパ
が必要
1. Busサイクル
2. Bus通信プロトコル
1. クロックサイクル
2. 通信と動作を一体化
B1
UnTimed
BCASH
BCA
FCA
B2
B3
B4

【図３１】



【書類名】　　要約書

【要約】

【課題】　論理装置の設計の初期段階において、動作シミュレーションを行なって、機能の確認，アーキテクチャの評価を行なえるようにするとともに、アーキテクチャの変更に対しても最小限の記述の変更で柔軟に対応できるようにする。

【解決手段】　スレッドマネージャ１１が、論理装置の動作完了までに必要となる機能を表したスレッド１３の実行に必要なハードウェアリソース１４をリソースマネージャ１２が要求に応じて割り当て、さらに、スレッドマネージャ１１がその割り当て結果に応じてスレッド１３の実行状態を制御するとともに、スレッド１３の実行が完了するまでスレッドマネージャ１１とリソースマネージャ１２とが連携してリソース要求，リソース割り当て及びスレッド制御を繰り返し実行することにより、論理装置の動作完了までの動作をシミュレーションする。

【選択図】　　図３

出　願　人　履　歴　情　報

識別番号　　〔０００００５２２３〕

１．変更年月日　　１９９６年　３月２６日
［変更理由］　住所変更
住　所　神奈川県川崎市中原区上小田中４丁目１番１号
氏　名　富士通株式会社